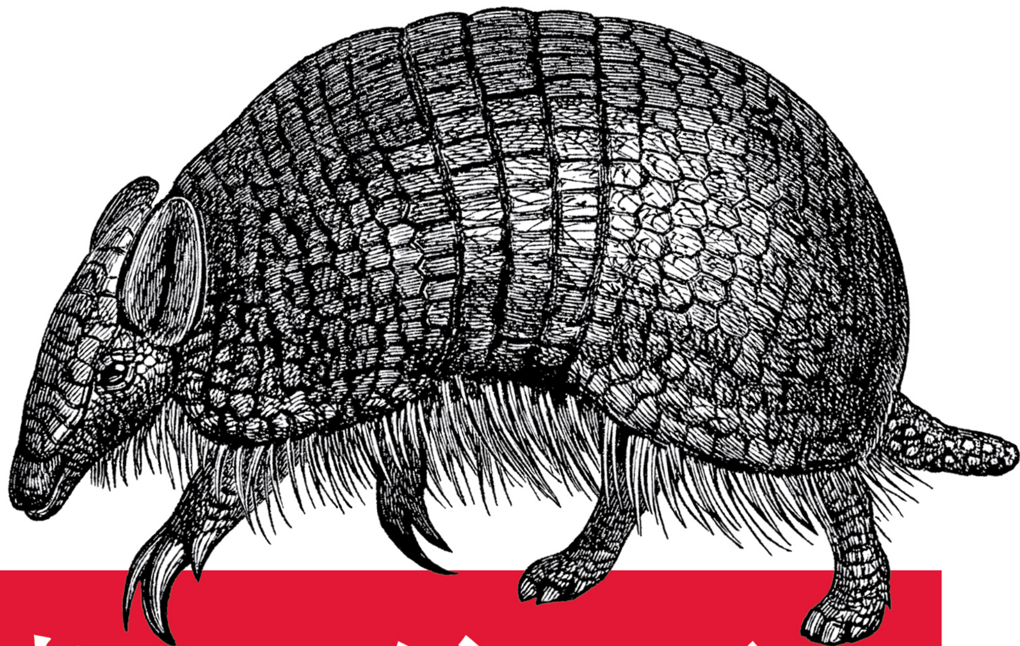
O'REILLY®

TURING 图灵程序设计丛书



数据压缩入门

Understanding Compression

谷歌开发高手通俗讲解数据压缩算法

高效传输和存储海量数据，打造流畅的用户体验

[美] 柯尔特·麦克安利斯　亚历克斯·海奇 著

王凌云 译

 中国工信出版集团　 人民邮电出版社 POSTS & TELECOM PRESS

译者介绍

王凌云

先后就读于大连理工大学与北京师范大学，现从事科技信息服务工作。阅读兴趣广泛，对数学、计算机、历史、文学等有浓厚的兴趣。除本书外，另译有《度量：一首献给数学的情歌》《软件开发本质论》。

数字版权声明


图灵程序设计丛书

数据压缩入门

Understanding Compression

Data Compression for Modern Developers

[美] 柯尔特·麦克安利斯　亚历克斯·海奇 著
王凌云 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社
北　京

图书在版编目（CIP）数据

数据压缩入门 / (美) 柯尔特•麦克安利斯
(Colt McAnlis) ，(美) 亚历克斯•海奇
(Aleks Haecky) 著 ; 王凌云译. -- 北京 : 人民邮电出
版社, 2020.4
（图灵程序设计丛书）
ISBN 978-7-115-53417-0

Ⅰ. ①数… Ⅱ. ①柯… ②亚… ③王… Ⅲ. ①数据压
缩 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2020)第027965号

内 容 提 要

本书的主题是数据压缩，也就是用紧凑的方式来表示数据。本书先讲解了 5 类数据压缩算法，即变长编码、统计压缩、字典编码、上下文模型和多上下文模型，然后介绍了香农的信息论，以及怎样通过各种方法来突破熵，如统计编码、自适应统计编码、字典转换、上下文数据转换、数据建模等。本书还讨论了数据压缩中的一些要点，如多媒体数据压缩和通用压缩，并介绍了有损数据压缩。本书最后说明了数据压缩与你、你的公司以及未来的技术是如何相互关联的。

本书适合对数据压缩感兴趣的开发人员阅读。

◆ 著　　[美] 柯尔特·麦克安利斯　亚历克斯·海奇
译　　王凌云
责任编辑　温　雪
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：13
字数：300千字　　　　2020年4月第1版
印数：1－3 500册　　　2020年4月北京第1次印刷

著作权合同登记号　图字：01-2019-8024号

定价：69.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O’Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O’Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

致 JAM 与 MLM：

我曾向祖鲁族（Zuul）发誓，如果你们不立刻把西兰花吃了，那么我就会写一本书，并且在献词中这样写道：你们连一种人类已经吃了几千年的菜叶子都害怕。等 20 年后你们也为人父母时，我再将这本书拿出来，让你们看我写下的话，并当面嘲笑你们。只有这样，你们才知道现在你们把我弄得多么抓狂。

致 KMKM：

让我们再试十年，怎么样？

柯尔特·麦克安利斯

致 AHS 与 GHS：

我希望你们能学会做饭。然而，你们证明了人类是可以靠吃新鲜的苹果和超市里卖的不新鲜的寿司存活的。

亚历克斯·海奇

目录

序

我第一次编程时，连数据压缩是什么都不知道，更谈不上认识到它的重要性了。幸运的是，我的 Apple II Plus 计算机内存有 0.000 048 GB（约 48 KB），这在 1979 年算是很大的内存了，足够让我去探索编程和计算机图形学。我当时完全没有意识到程序和数据在后台是不断压缩和解压缩的，从而减小了它们在内存中的大小。说到这里，真要感谢 Woz！

有了几年的编程经验后，我发现：

- 数据压缩需要花费时间并可能会导致软件变慢；
- 改变数据的组织结构可以让数据压缩得更小；
- 复杂的数据压缩算法各式各样。

这使我意识到压缩不是一个刚性的黑盒，相反，它是一个灵活的工具，可以极大地影响软件的质量。我们可以通过以下几种方式运用压缩：

- 改变压缩算法有可能让软件运行得更快；
- 针对数据的组织结构选择正确的压缩算法，可以使数据变得更小；
- 选择不匹配的数据组织结构或压缩算法，可能会导致数据变大或运行变慢。

如今我明白数据压缩为什么重要了。如果数据太大、内存不够，或者解压缩太慢，可以稍微改变一下数据的组织结构，以使它更好地适应压缩算法。比如，我会将数单独放在一组，字符串放在另一组，为重复出现的数据类型建表，或者将分数截断为整数。如果能让数据与算法匹配，我就无须去做评估与采用新的压缩算法这样的苦差。

后来，我开始做专业的电子游戏，大部分游戏数据是由艺术家、设计师和音乐家这样的非技术人员生成的。结果表明，数学不是他们最喜欢讨论的话题，而且他们对改变游戏数据以便更好地利用我的单向压缩算法不太感兴趣。好吧，既然数据的组织结构无法改善，剩下能做的就是选择最合适的压缩算法来与这些“伟大的”艺术数据匹配了。

我调查了各种数据压缩算法，发现有两大类很适合电子游戏数据：

- **无损方法**
  - 去掉重复数据（LZ 算法）
  - 熵压缩（哈夫曼编码、算术编码）
- **有损方法**
  - 降低精度（截断或降采样）
  - 图像 / 视频压缩
  - 音频压缩

对文本字符串和二进制数据使用 LZ 算法，可以将完全重复的数据压缩掉。对像素数据使用有损的矢量量化（vector quantization，VQ）算法，可以将像素映射为调色板。对音频数据使用有损的降采样和线性预测编码（linear predictive coding，LPC）算法，可以减少每秒的二进制位数。如果 CPU 足够快，前述所有这些压缩算法的输出都可以再用无损的哈夫曼算法进行一次额外的统计熵压缩。

20 世纪八九十年代，我参与制作了大约 30 个游戏，其中大多数使用的是这些算法，外加简单的数据构造工具对数据的组织结构进行有限的优化。

但是到了 2000 年左右，情况变得复杂起来。数据生成工具与数据展示和分析工具之间展开了持续的竞争。其结果是软件性能、存储大小、网络拥塞，以及压缩算法与数据组织结构的有效配对。

这种数据洪流被更大的存储空间（蓝光光盘、TB 级的硬盘以及云存储）、更快的多核 CPU、新的无损压缩算法（如 BWT、ANS 和 PAQ），以及针对图像、视频、音频等数据的有损编解码器的巨大性能提升部分抵消了。然而，由于数据每年的增长速度太快，相比之下，网络带宽的增加、压缩算法的性能提升以及存储容量增长的速度就太慢了。

这些因素造成了我们的现状，这也是本书内容之所以重要的原因。

程序员怎么才能知道为数据选择哪种压缩算法，以及对数据做什么样的改变能使特定的算法表现得更好或更差呢？其实真正有帮助的是对主要的数据压缩算法进行介绍，指导开发人员从众多可用的算法中选择最合适的。大部分开发人员其实并不需要掌握实现这些算法所需要的所有理论和数学细节，他们只要知道这些算法的优缺点，以及在特定场景中怎样充分利用它们即可。

我很高兴在过去的 37 年里一直在实现和使用不同的数据压缩算法，并看到它们的发展变化。我希望这本书能揭开数据压缩的神秘面纱，为软件开发人员学习压缩算法提供一个起点，同时帮助他们开发出更好的软件。

John Brooks
Blue Shift 公司首席技术官

前言

数据压缩无处不在，对现代计算来说它仍然像以前一样必不可少。过去，1 GB 就已经很大了，数据以每秒几千字节的速度传输。从某种意义上来说，我们已经经历了一个完整的循环，从内存和带宽有限的古董计算机时代，来到了内存有限且数据套餐十分昂贵的移动设备时代。

幸运的是，有很多工具、API 以及程序包可以帮我们压缩数据。理解它们如何工作，有助于我们**正确**地选择压缩工具（或算法），而这又可以令用户更高兴，同时降低成本、增加收入。

数据压缩的基础是数学，让我们坦然面对它。对大多数人来说，数学很难，真的很难，而且对于程序员曾经是最高的一道门槛。想想数据压缩之父克劳德·香农（Claude Shannon），他的数学非常好，在黑板上随手一写就是一行行复杂的方程。

更疯狂的是，现代程序员不需要了解数学。现在，8 岁的孩子都能上网，甚至在没有上过代数课的情况下，就能通过自学教程发布自己的网页或应用程序。

我们相信，这就是过去 20 多年里数据压缩领域一直停滞不前的原因。虽然有 20 亿人在使用移动设备[1]，并且他们经常遇到内存不足和网络连接不良等问题，但是数据压缩技术仍然处于半停滞状态。这是因为懂数学的程序员不多。

当然也因为数学比较难。

你可以看到，压缩不是真的与数据有关。数据压缩领域早期的创始人考虑的并不是数据，而是统计。他们寻找并发现了操纵数据集中符号的**概率分布**的不同方法，并利用这些方法来生成包含同样的信息但更小的数据集。

注 1：这是 2015 年的数据，如果你是在未来某个时间点读到这本书的，数据肯定会不同。还有，很感谢你阅读这本书。

随着计算机技术越来越普遍、越来越去数学化，普通程序员需要知道的统计学知识和其他高等数学知识也越来越少。因此，尽管 21 世纪初出现了计算机史上最大规模的技术繁荣，**整个数据压缩领域**却只取得了两三项技术进展。

因为数据压缩很难。

因为它以数学为基础。

现在，我们从公平和实用的角度来看待这个问题。如今，大多数程序员和内容开发人员不需要懂得高等数学，也不需要理解压缩的工作原理，因为他们只需要获得一个像样的数据压缩库，再把数据扔给它，就可以到处使用压缩后的数据了。

然而，向前看，这还不够。根据预测，到 2025 年，将有 50 **亿**人使用计算机并通过互联网传输数据。想想那时，数据量会急剧增长，我们会有太多的数据，运营商的传输速度会不够快，数据仓库又太小而无法容纳这些数据。当然，一个解决方法就是使用尚未发明出来的创新算法，实现更快、更好的压缩。

这自然要用到数学。

而数学又很难。

另一个解决方法是教那些愿意学的人理解数据压缩的工作原理。因此，你不再是随便拿到某个压缩工具就去使用，而是可以选择最好的压缩工具，并将数据以最高效的方式提供给用户。

这就是本书的写作动机。我们试图将数据压缩这一学科中大量难以理解的内容简化为普通人都能理解的内容，并且让他们能将这些知识应用到日常的数据需求中。我们试着尽可能少用数学，尽量用图、表和数据流的形式来解释数据压缩的基本原理。与柯尔特在 YouTube 上的 *Compressor Head* 系列视频相似，我们希望能通过本书教给任何高中以上文化水平的读者一些数据压缩知识，即使你不是程序员也不要紧。

不过，我们要坦诚地告诉读者：如果你真想理解这些内容，就必须做一些思维训练。就像骑自行车一样，数据压缩还是比较难的，只有你真正领会了其中的要点，一切才变得有意义起来。但在这之前，你必须坚持下去，并通过例子加深自己的理解。

有一点需要明确，本书的目标不是让你成为压缩专家，因为那需要相当深厚的数学功底（这很难做到），而是让你理解压缩**算法**，这意味着我们有时会使用恰当的术语，有时则会使用虽然不太正确但更具描述性的术语。掌握这些术语也许还不足以让你与其他数据压缩专业人员在茶歇时随心交流。我们想为你提供足够多的信息，以确保你做出正确的有关数据压缩的商业决策。

最后，老实说，数据压缩真的**很酷**。当然，这是我们的想法，希望当你深入阅读本书后，也会这样认为。

在写这本书时，我们觉得很开心，也希望你能从数据压缩知识的学习中获得乐趣。

怎样阅读本书

与任何好故事一样，本书也回答了所有相关的疑问。**什么**是数据压缩？**为什么**需要了解数据压缩？数据压缩这一技术是**什么时候**发明的？有**哪些人**致力于减少更多的二进制位？我们需要在产品开发周期的**哪个阶段**关注数据的大小？还有最重要的，它是**如何**节省二进制位、金钱以及用户的数据流量的？

本书是按先后顺序编排内容的，因此我们建议你从前往后按章阅读。本书的每一章都以前一章为基础，这不仅体现在时间线上，也体现在术语的引入和算法的演化方面。我们是根据从前往后阅读的设想写作本书的，因此按照我们设想的方式来阅读最简单。

怎样从后往前阅读本书

如果能让你激动的是钱而不是复杂的算法，你可以试着从后往前倒着读这本书。要让你自己完全相信，数据压缩是自（蘸黄油）切片面包发明以来最令人激动的事情；然后在这一信念的鼓舞下，再去理解数据压缩是如何工作的（你猜对了，理解了数据压缩之后，就可以赚更多的钱）。准备好了吗？

章节概要

不得不说，这一部分内容通常很无趣，但我们向你保证，就像电影《公主新娘》中的原作者及编剧 William Goldman 那样，我们只向你展示干货。因此，虽然我们要求你不要跳过第 2 章，但你完全可以跳过这个概要。你可以通过浏览目录来了解整本书的内容，或者既然你已经拿起本书，那就干脆直接读下去。但如果你想在实际阅读前了解一下各章的内容，这里会给你一些提示。

- **第 1 章，并非无趣的一章**

  如果你没有时间阅读整本书的话，那么阅读第 1 章就可以了，这里有你需要知道的所有内容。压缩算法可以分为 5 类：变长编码、统计压缩、字典编码、上下文模型和多上下文模型。香农发明了一种度量消息中所包含信息内容的方法，并称之为信息熵。压缩的关键在于使用尽可能少的符号对数据进行编码，以使其占用的二进制位数尽可能地少。数据压缩是整个互联网的基础，我们应该对所有的数据进行压缩。这就是第 1 章的内容，如果你愿意，现在就可以合上这本书了。

- **第 2 章，不容错过的一章**

  这一章之所以不容错过，是因为它奠定了全书的基础，比如如何用 0 和 1 来表示整个世界，同时还介绍了信息论和熵的概念（即表示一个数所需要的最小二进制位数）。

- **第 3 章，突破熵**

  根据香农的理论，熵就是一组数据在理论上所能达到的最小大小，而数据压缩则通过利用真实数据的两个性质，即符号之间的顺序与相互关系，打破了这个限制。

- **第 4 章，VLC**

  在这一章中，你会学习如何将 0 和 1 串在一起，构造唯一的、长度可变的码字，然后再将最短的码字分配给数据集中最可能出现的符号。此外，你还会遇见 Peter Elias。

- **第 5 章，统计编码**

  不存在适用于所有场合的万能方法，而统计编码算法能根据特定的数据集进行优化，生成个性化的变长编码。在这一章中，你将使用便签构建哈夫曼树，并学习算术编码，还会遇见 Jarek Duda，他通过引进算术数字系统取代了前两者。

- **第 6 章，自适应统计编码**

  真实的数据流时刻在变化，而自适应编码器能根据正在处理的数据的局部特性对编码进行优化。

- **第 7 章，字典转换**

  如果不能进一步压缩数据，可以考虑在应用统计压缩之前，以“单词”为单位，对其在数据集中的重复方式进行编码，从而将数据预处理为更可压缩的形式。

- **第 8 章，上下文数据转换**

  有多少数据集，就有多少不同的转换，但下面两种转换是现代计算中最为重要的：行程编码和增量编码。这两种方法的共同之处在于，它们会根据前面已经出现过的内容来决定后面会出现的内容。

- **第 9 章，数据建模**

  多上下文编码算法为了识别编码当前符号的理想的二进制位数，会考虑最后见到的几个符号。你可以将它看成每读取一个新的符号就去创建一棵新的哈夫曼树。在这一章，你会遇到如下术语：马尔可夫链、部分匹配预测算法和单词查找树。如果想给朋友留下深刻印象，不妨用用上面这些术语。

- **第 10 章，换个话题**

  这一章简单介绍多媒体数据压缩，并为数据的实际使用打开方便之门。

- **第 11 章，评价数据压缩**

  针对不同的数据流有不同的数据压缩算法。使用场景不同，使用的压缩算法也会不同。在这一章中，你会学习为了选择最符合当前需要的压缩算法需要考虑哪些方面。

- **第 12 章，压缩图像数据**

  如果你是一名应用程序开发人员，那么需要处理的绝大多数数据会是图像。我们可以利用规律并巧妙地忽略那些人类大脑注意不到的信息，来节省图像需要的二进制位数。在这一章中，你将学习 PNG、JPG、GIF 和 WebP 等图像压缩格式。

- **第 13 章，序列化数据**

  在网络应用程序中，序列化的内容可以说是第二常见的数据传输格式。我们可以用二进制序列化格式来代替 JSON 和 XML 格式。

- **第 14 章，有损数据压缩**

  实际上，如果深入讨论这个话题的话，可以写成另外一本书。

- **第 15 章，让世界变得更小**

  这一章会回答，为了让客户更多、事业更兴、利润更高，你必须关注数据压缩的原因。

电子书

扫描如下二维码，即可购买本书中文版电子版。



第 1 章

并非无趣的一章

亲爱的读者，欢迎来到本书第 1 章，本书要讨论的主题在数据处理中有着重要的地位。这一章将为你阅读本书后续内容打下基础（出版者这么说），并激发你的阅读兴趣。为了帮助你以尽可能轻松平缓（又有趣）的方式进入数据压缩这一领域，出版者希望我们先介绍一下数据压缩的历史、一些基础知识，还有其他任何我们能想到的内容——但不涉及数学，因为数学很难[1]。

但是说实话，如果真那样做的话，不仅读者读起来无趣，作者写的时候也没意思。

所以，我们决定不那样做，而是根据自己的想法来写。本书的主题是数据压缩，而数据压缩无非是用最紧凑的方式来表示数据。因此，我们会用最简短的篇幅来写这介绍性的第 1 章。

这一章首先介绍数据压缩算法，然后介绍克劳德·香农。他几乎毁了我们的生活，同时又创造了我们所热爱的计算机的几乎所有重要方面。最后，这一章会介绍关于数据压缩你必须知道的一件事。我们相信，通过这种方法，可以让你弄清楚数据压缩怎样使应用更好、更便宜、更快。

亲爱的读者，就这样说定了，好吗？

1.1　5类数据压缩算法

数据压缩算法其实是一个相当大的主题，好在我们可以对所有这些算法进行分类。分类之后，这些算法理解起来也就更容易了。简单地说，数据压缩算法有 5 类：**变长编码**

注 1：这是我们最后一次这样说。

（variable-length codes，VLC）、**统计压缩**（statistical compression）、**字典编码**（dictionary encodings）、**上下文模型**（context modeling）和**多上下文模型**（multicontext modeling）。所有这 5 类算法都有很多变种，这是一件好事，因为我们可以根据自己的需要来选择。每类算法的变种在输入数据、算法性能、内存要求以及输出大小方面存在细微的差别。要选出其中最佳的一个算法，需要在准备的数据上测试这些算法，然后找出压缩效果最好的那个。

这几类算法也可以混合使用，因为其中有些算法的目的就是转换数据，使得其他的算法在压缩时更有效。

如果想成为数据压缩方面的专家，那么你就必须理解这几类算法、如何搭配使用它们，以及对于特定的数据集需要从哪类算法中选出一个具体的算法。

下面我们开启学习之旅吧。

1.2 惹人“愤怒”的克劳德·香农

让我们先回到 20 世纪 40 年代，一位名叫克劳德·香农的统计学家发表了好几篇论文，详述了他在第二次世界大战期间在军队任职时以及之后在贝尔实验室工作时所做的研究工作。

香农非常聪明，数学学得特别好。在 1936 年离开密歇根大学之前，他已经取得了工程技术学士学位和数学学士学位。随后，他又去了麻省理工学院，并在那里做了很多研究工作。他的硕士论文题为《继电器与开关电路的符号分析》，该论文为基于开关的现代电路计算奠定了基础。

1948 年，香农又发表了《通信的数学理论》，在这篇论文中他详细论述了发送者怎样对要发送的信息进行编码才能达到最佳效果，由此开创了**信息论**（information theory）这一全新的学术领域。对消息进行编码有多种方式，“字母表”与“摩尔斯码”只是其中常见的两种。但是对每一个特定的消息来说，都有一个最佳的编码方式，这里的“最佳”指的是传递消息时用到的字母或者符号（也可以说是二进制位，即信息的单位）最少。至于这里说的“最少”到底是多少，则取决于消息所包含的**信息内容**。香农发明了一种度量消息所携带信息内容的方法，并称之为**信息熵**（information entropy）。

数据压缩其实是香农的研究工作的一项实际应用，它所研究的问题是，“在保证信息能恢复的前提下，我们能将消息变得多么紧凑”。[2]

不过，等一等，为什么我们要在这一节的标题中说香农惹人“愤怒”呢？

注 2：需要注意的是，根据现代信息论的观点，在压缩数据以减少总二进制位数的时候存在一个临界点，如果超过了这个值，我们就不能将压缩后的数据唯一正确地恢复为原来的数据流。因此，我们的压缩目标就是尽可能地减少总二进制位数以接近这个临界值，并且不超过这个值。

答案是，虽然我们要感谢香农帮助创造了现代计算机——这本书就是在计算机上写作后出版的（你也很可能是在计算机上阅读这本书的），但他在信息论方面的工作一直以来都是我们想要突破和超越的。你可以把数据压缩看成对信息熵的挑战。计算机科学家所写的每一个数据压缩算法都是为了反驳香农的研究，**使数据的压缩程度超过用香农发明的公式计算出来的信息熵**。我们想方设法地去掉信息中每一个冗余的二进制位，想让它变得尽可能小，以突破香农所定义的熵的下限，从而达到对信息的全新层次的理解。在过去的 60 多年里，工程人员花了大量时间，想通过创造新的算法来超越或者巧妙地绕开这位伟人所创造的概念。

1.3 关于数据压缩，你必须知道的

下面直接上干货。

对数据进行压缩，通常有两个思路：

- 减少数据中不同符号的数量（即让“字母表”尽可能小）；
- 用更少的位数对更常见的符号进行编码（即最常见的“字母”所用的位数最少）。

好了，你需要知道的就是这些。

60 年来的数据压缩研究都可以归结到上面两个重要思路上，数据压缩中的每一个算法都聚焦于解决这两件事情中的一件。每一个压缩算法，要么通过打乱符号或减少符号的数量，将数据转换得更便于压缩；要么利用其中一些符号比其他符号更常见的事实，通过用最少的位数编码最常见的符号，实现压缩的目的。

虽然数据压缩的思路简单明了，但在实际应用中数据压缩很复杂。其原因在于，由于要压缩的数据的类型不同，针对上述两条思路中的每一条，能采用的方法都有很多。因此，进行实际的数据压缩时，需要综合考虑以下些因素。

- 不同数据的处理方法不同，比如压缩一本书中的文字和压缩浮点型的数，其对应的算法就大不相同。
- 有些数据必须经过转换才能变得更容易压缩。
- 数据可能是偏态的。例如，夏天的整体气温偏高，也就是说高气温出现的频率比接近零度的气温出现的频率高很多。

作为程序员，你面对的挑战就是，找出最好的方法或者方法组合来压缩用户给你的数据。而作为内容开发者，你面对的问题则是，如何在用户可接受的费用范围内将数据传递给用户。[3]

注 3：这是因为在世界上的大多数地方，数据服务套餐是按流量计费的，并且不便宜。

亲爱的读者，这就是本书接下来的全部内容。它将作为指导手册，让你知道在数据压缩这一领域内有哪些知识值得关注，让你理解压缩算法**理论上**是怎样工作的，以便你能从中选择最适合的算法，并将它应用到你开发的那些酷炫的社交 / 移动 / 网络 / 多媒体应用程序数据上。

建立在数据压缩上的世界

我们需要知道这样一件事：我们当下生活在其中的这个计算世界，完全建立在数据压缩算法之上。

是的，每个部分都是如此。

每个网页、每个图像、每首歌、每个关于猫的视频、每个流媒体网络电影、每张自拍照、每次电子游戏下载、每个微型交易，甚至是操作系统的每次更新，所有这一切都得益于压缩算法。事实上，哪怕只是想通过互联网传输一个二进制位的数据，也离不开压缩的内容。

数据压缩技术最让人惊异之处在于，它与过去 40 年里个人计算的很多重大改变有关，但很少有人知道这一点。

例如，你是不是通常不买 CD 唱片，而是直接下载或在线听音乐？如果的确如此，那你需要感谢压缩算法。

1. 音乐的压缩

时间回到 1996 年，一群来自不同公司的聪明人组成了一个联合工作组，推出了 MP3 这种文件格式。这种新的音频格式从此改变了计算机中音频的特性。当时，WAV 格式才是创建、存储和传输音频数据的主流格式。几乎所有人在用 WAV 格式，但它存在一个很严重的问题，那就是文件特别大。一首 3 分钟的歌曲，文件的大小就将近 30 MB，下载要花费 9 分钟左右，更别提流式播放了。[4]

MP3 格式出现之后，一首音频质量很好、3 分钟左右的完整歌曲，文件大小只有 1~3 MB[5]。人们甚至可以将 CD 唱片放入计算机，并将其转换成 MP3 格式，以便以数字信号的形式来欣赏。

较小的文件大小与较高的音频质量，这两个优点相结合产生了我们这个时代最伟大的消费创新之一：Napster 音乐共享平台。这项服务让音乐爱好者免费交换 MP3 文件成为可能。然而，由此也产生了一个很大的法律问题，那就是一些人在买了 CD 唱片之后将其转换为

注 4：如果感兴趣的话，可以阅读 *The Web Back in 1996–1997* 一文了解当年走过的弯路。

注 5：需要注意的是，MP3 是一种有损数据压缩格式，也就是说在压缩过程中会有一些信息丢失。后面的章节中会简单地讨论这种数据压缩类型。

MP3，然后与朋友共享，这样他的朋友就可以免费听这些唱片了。如此一来，最终的结果你也能想象到，这种行为动了 CD 发行公司的奶酪，因而遭到了这些公司的强烈反对，他们动用了一切手段，最终成功地让 Napster 音乐共享平台关闭。

在 20 世纪 90 年代末至 21 世纪初，类似的法律纠纷很多，政府的政策也不断变化，试图阻止这种形式的音乐共享。甚至有人提出“使用 MP3 格式是非法的”的立法建议。

苹果公司没有参与打击这种新的数据格式，而是决定围绕它生产一个产品。1998 年，苹果公司推出了 iPod，这是最早的专门存储和播放 MP3 文件的便携式设备之一。随后，苹果公司又推出了 iTunes 商店，让顾客可以合法购买 MP3 文件供个人使用[6]。

今天，数字音乐的发行已经成为新常态，很多公司尝试找到更好的方式来促进音乐的营销。

iPod 这一产品的巨大成功，最终带来了 iPhone 的开发和发布，由此永远地改变了个人计算的面貌。（这是另外一个故事了。）

2. 图像的压缩

让我们把时间再往回倒一些，回到互联网诞生的那一年，也就是 1978 年。当首批互联网连接建立起来的时候，能发送的数据量非常少。当时的用户也很少，网络主要用来发送和接收文本数据，或者如图 1-1 所示的完全用字符创建的图像[7]。

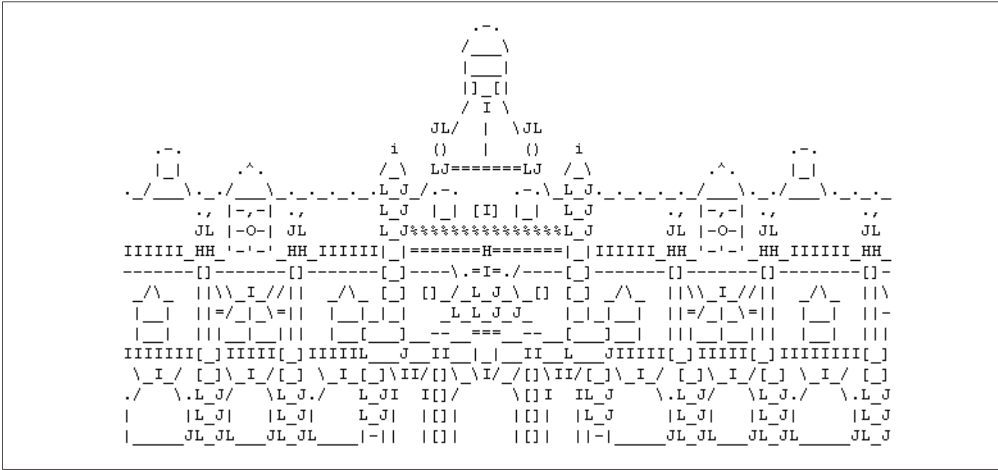


图 1-1：由 ASCII 字符画出来的城堡，来源：维基百科，佚名作者

当时的问题是，真实的图像信息以 24 位 / 像素格式存储，对早期的调制解调器来说这样的数据量实在是太大了。因此，压缩专家将图像压缩设为目标。为了测试新的图像压缩算

注 6：第一个便携式 MP3 播放器是由世韩信息系统公司于 1997 年推出的，而首先推出流媒体服务的则是美国电话电报公司。

注 7：ASCII 艺术实际上是由一些有创意的人使用打印机发明的。

法，他们需要一个图像语料库。在一个男性主导的行业中，他们更偏向于从男性杂志中寻找图片素材，最终选择了现在很著名的莱娜图（见图 1-2），该图截取了 1972 年 11 月《花花公子》杂志中莱娜·瑟德贝里（Lena Söderberg）的一张照片的一部分。

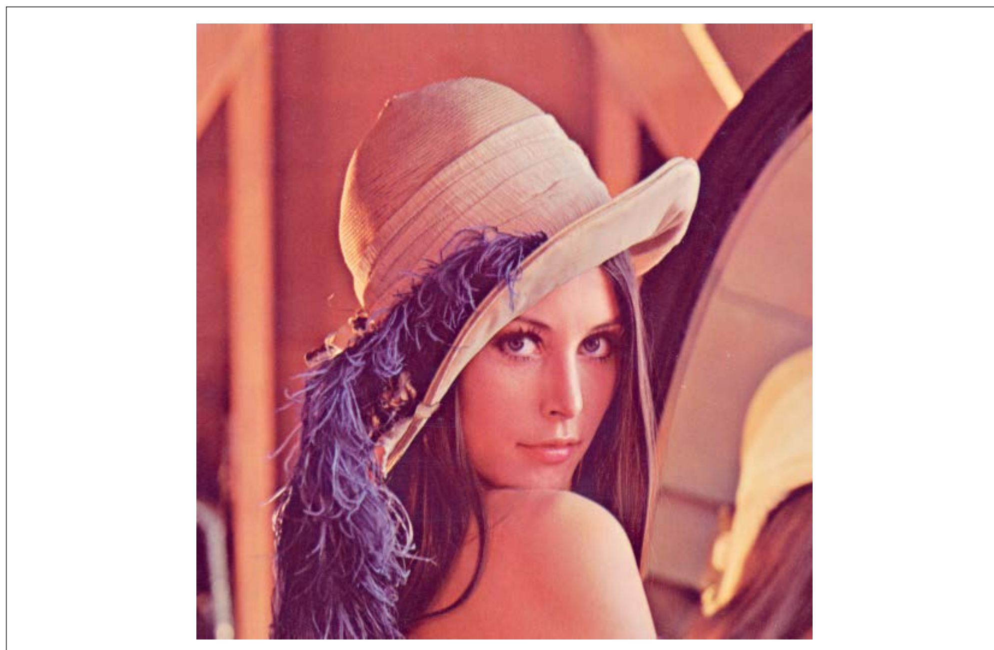


图 1-2：莱娜图，这幅图像的原始全身照是由 Dwight Hooker 拍摄的，并发表于 1972 年 11 月《花花公子》杂志的“当月玩伴”栏目。这幅 512 × 512 的图像是由 Alexander Sawchuk 等人通过电子或机械扫描原始照片的一部分获得的，现在可以从 USC-SIPI 图像数据库中获得。该图片已获得维基百科授权

在发表研究成果时，他们在论文中使用了裁剪后的 PG-13 版本的图像，并提供了原始的图像版本以供其他研究人员测试压缩算法。在很长一段时间内，莱娜图是用来测试图像压缩算法的标准测试图。幸运的是，从此以后，很少再出现这样有争议性的图像语料库。（我们两位作者则更喜欢使用柯达公司的图像测试集。）然而即使是今天，仍然有很多图像压缩方面的论文将莱娜图用作检验算法的标准。

3. 视频的压缩

让我们快速回到 2001 年，这一年 YouTube 网站出现了，在这个网站上用户可以免费上传他们录制的任何视频，供所有人观看。

这时，视频信息的主流传输格式是 MOV，然而因为这一格式不比将一系列 JPG 图像按顺序排列先进，所以相应的文件很大也就不足为奇了。因此，只加载网页就能观看视频的想法在当时实在是令人难以置信。

4. 基因图谱

2008 年，为了治疗人类疾病，降低疾病死亡率，科学家开始绘制和测试人类基因组。单个基因组序列就包含了大量的数据，仅仅是描述人类基因组成的数据就超过了 14 GB。这样的数据大小超出了大多数系统能处理的范围（当时云计算还不是热门话题）。

数据压缩再一次成为解决问题的利器。研究人员发现，BWT[8] 是最有效的存储 DNA 信息的压缩格式，甚至无须解压就能对数据进行操作。

到了 2014 年，研究人员通过将可扩展的云计算与主机之间的压缩数据传输结合起来，创造了全球最快的蛋白质折叠计算之一。

5. 压缩与经济

通过前面的介绍，我们可以看到，数据压缩一直是推动计算技术与计算文化发生重大变化的核心力量。这背后的经济原理却很简单：压缩后的文件会变得更小。这也就意味着，同样的数据传输所需的时间会变短，相应的费用也会减少。分发者的分发成本会降低，消费者的支出也会减少。在当今这个计算时间就是金钱的时代，数据压缩可以说是缩短内容分发者和消费者之间距离最经济可行的方法。

注 8：第 8 章会对 BWT（Burrows-Wheeler transform，伯罗斯－惠勒变换）进行更深入的讨论。

第 2 章

不容错过的一章

即使你对**二进制数**很熟悉，**本章的内容同样不容错过**。本章还将开始深入研究**信息论**，这是理解本书后面内容的前提。

2.1　理解二进制

这看起来或许有些奇怪，一本论述数据压缩的书居然以二进制数开头。还请读者多担待，因为数据压缩所做的无非就是尽可能减少表示特定数据集时所需的二进制位数量。为了进一步阐述这一概念以及它在数学上的影响，我们不妨在此花一点时间，确保每个人的理解程度一致。

2.1.1　十进制计数系统

现代数学建立在十进制计数系统之上。[1]

有了这一计数系统，我们就可以用 [0,1,2,3,4,5,6,7,8,9] 这 10 个数字来表示任何数值。上小学时，你可能已接触过数位的概念，比如 193 这个数从左到右包含 3 个数位，分别是百位、十位和个位，如下表所示。

百位	十位	个位
1	9	3

注 1：这里说的是现在的情况，我们相信未来某一天量子计算或巴比伦计数法会改变这一点。

实际上，193 就等于 1 × 100 ＋ 9 × 10 ＋ 3。一旦掌握了这个规则，你就会认识到可以一直这样数下去，数到任何数都可以。

后来，学了指数后，你知道可以用以 10 为底的指数去代替百位和十位，这样新的规律又出现了，如下表所示。

10^2	10^1	10^0
1	9	3

因此，下式成立：

$$193=1\times100+9\times10+3=(1\times10^2)+(9\times10^1)+(3\times10^0)$$

因为每一列只能有一位数字，所以在 9 的基础上再加 1 会发生什么呢？从 9 再数一位，得到 10（两位数）。因此，将 0 放在个位（10^0）列，并将 1 向左移一位，也就是移到 10^1（十位）列，该列表示的是十位，如下图所示。

10^2	10^1	10^0
		9

+1

10^2	10^1	10^0
		10

10^2	10^1	10^0
	1	0

如果接着往上数，我们会遇到 19 ＋ 1=20（2×10^1）。当数到 99 时，再往上数，就需要进位继续左移，因此结果为 1×10^2。

2.1.2 二进制计数系统

二进制计数系统的工作原理与十进制计数系统一样，唯一的区别是前者的基数为 2，而后者的基数为 10。因此，与十进制中每一列都表示为以 10 为底的指数（10^2 | 10^1 | 10^0）不同，在二进制中每一列都是以 2 为底的指数（2^2|2^1|2^0）。

此外，十进制中在进位之前有 0~9 这 10 个数字可用，而在二进制中只能使用 0、1 这两个数字。

因此，在二进制中计数，我们有“0”“1”，同时由于 2^1 已进位到了下一位，因此二进制中“2”表示为 10，“3”表示为 11，而“4”等于 2^2，需要再进一位，因此表示为 100，如下图所示。

2^2	2^1	2^0
		1

+1

2^2	2^1	2^0
		10

2^2	2^1	2^0
	1	0

1. 将二进制转换为十进制

当你阅读本章前面的内容时，我们确信你的大脑已经将二进制数转换成了相应的十进制数，这是因为除非一直以来都使用二进制数，否则你还是会通过其对应的十进制数来理解二进制数。

让我们更明确一些，假定有二进制数 1010，将它填到各数位的列中，如下表所示。

2^3	2^2	2^1	2^0
1	0	1	0

为了得到对应的十进制数，我们将数值为 1 的列对应的值加起来，通过上面的表格，得到下式：

$$2^3 + 2^1 = 8 + 2 = 10$$

因此，二进制数 1010 等于十进制数 10。

通过上面的步骤，可以看到，将二进制数转换为十进制数很简单。反过来，将十进制数转换为二进制数则要稍微复杂一些。

2. 将十进制转换为二进制

要将十进制数转换为相应的二进制数，一种简单的方法是用十进制数一直除以 2，所得的余数为“1”或“0”，然后将所有这些余数串起来。

实际操作一次更便于理解。下面就用这种方法将十进制数 294 转换为相应的二进制数。

(1) 首先用 294 除以 2，所得的商为 147，余数为 0。
(2) 将步骤 (1) 中所得的商 147 继续除以 2，此时商为 73，余数为 1。
(3) 将步骤 (2) 中所得的商 73 继续除以 2，得到相应的商和余数，重复这样的操作，直到所得的商等于 0 而余数等于 1，由此得到下表。

用十进制数重复除以2 294	列对应的值		
147	余数	0（LSB）	2^0
73	余数	1	2^1
36	余数	1	2^2
18	余数	0	2^3
9	余数	0	2^4
4	余数	1	2^5
2	余数	0	2^6
1	余数	0	2^7
0	余数	1（MSB）	2^8

需要注意的是，如果上述步骤中被除的十进制数是偶数，那么该十进制数将被 2 整除，余数为 0；而如果该数是奇数，那么该数将不能被 2 整除，余数为 1。

现在将所得的余数从右向左串起来（通过上表来看是从下往上串起来的），将**最低有效位**（least significant bit，LSB）放在最右边，**最高有效位**（most significant bit，MSB）放在最左边，由此得到 100100110。

就这样，通过一直除以 2 这种将十进制转换为二进制数的技术，我们得到了最终的结果 100100110，它就是与十进制数 294 对应的二进制数。

一通百通，一懂百懂

结果表明，这种除以基数的方法也可以将十进制数转换为其他进制数。在计算机科学领域，另外一种常用的基数是 16，也称为十六进制数。由于没有哪个一位的数字符号能表示 10 以上的十进制数，因此我们用字母 A 来表示 10，用 B 表示 11，以此类推，用 F 表示 15。读者不妨试一试将十进制数 3053 转换为十六进制：先除以 16，再将所得余数从右向左串起来。友情提示：转换的结果可能会让人产生睡意。

2.2 信息论

好了，既然现在所有的读者都掌握了二进制数，处于同一水平，那么下面就来讨论在**信息论**的背景下二进制意味着什么。

信息论（名词）

> 即从数学的角度研究如何利用符号序列、脉冲序列或其他形式对信息进行编码，以及信息能以多快的速度在计算机电路或者电信信道中传输。

根据信息论的观点，一个数值所包含的信息内容等于，为了在一个集合中唯一地确定这个数值，需要做出的二选一（是 / 否）决定的次数。

每个孩子都是应用信息论的专家

20 个问题（20 Questions）这一游戏很好地阐释了信息内容这个概念。这种游戏的玩法是，第一个玩家随便在心里想一个东西，第二个（或其他）玩家通过提问来猜出这件东西，后者最多可提问 20 次。第一个玩家针对每次提问都用“是”或“否”来回答。

鉴于参加游戏的是小孩，我们会对这个游戏进行一些简单的修改，比如限定第一个玩家所想东西的范围（例如只能是动物），或者只有在得到了 10 次“否”的回答后，游戏才算结束（如果第二个玩家很快就猜出答案的话，第一个玩家可以撒谎或者改变所想的东西，这些都是允许的）。

这个游戏表明，要确定任意一个东西是什么，所需的信息量最多是 20 个二进制位。从数学的角度来看，如果所问的每一个问题都能排除一半的东西，那么 20 个问题事实上可以让提问者区分 2^{20}（即 1048576）个东西。

这真是很多很多的东西了。

我们还可以更进一步。

考虑如下场景：有一个 100 平方英尺[2]的房间，地上铺了 100 块瓷砖，上面是拱形的天花板，靠近东面的窗户边有一张四柱大床，北面的墙边有一张古色古香的小写字台，而在床的西边则放着一个沉重的 18 世纪的衣橱。

你完全可以在一页纸上用 JSON 或者其他你喜欢的脚本语言写出上面所有这些内容。

或者，你也可以采用下面这种略有不同的方法，用 7 个二进制位对地上的 100 块瓷砖进行编码，再用 2 个二进制位对 4 个主要方向进行编码，然后对 3 件家具进行编码，每件家具各用 2 个二进制位。按照瓷砖 – 家具 – 方向这样的顺序，你睡的床可以用 10010100111 来表示。（每个二进制位的“意义”这样的元信息，可以存在你的头脑中，或者编码到组装这个房间的软件中，抑或附在这些数据的后面。）

现在，描述整个房间只需要 44 个二进制位或者 6 个字节就够了，比起文字描述或者 JSON 文件，节省了相当多的空间（不仅我们这样认为，那些下载了你的游戏的移动端用户也会认同）。

简单来说，这就是一个数据压缩的过程。

注 2：1 英尺约合 30.48 厘米。——编者注

2.2.1 二分查找

假定有一个已经确定了范围且排好序的整数数组，比如说 0~15，我们想找出数值 10 在这个数组中的位置。

二分查找算法的工作原理是这样的：首先将数组中的数据集分成两半，然后判断要找的数值 10 比处于中间位置的枢轴值是大还是小[3]。根据对比的结果，我们将数组分成了两半，并保留包含 10 的那一半数据进行进一步的查找。然后再次将 10 与新的枢轴值进行比较，并继续将数组分成两半。一直这样进行下去，直到最终只剩下我们要找的数值 10。（如果这听上去与前面所说的 **20 个问题**游戏有些相似，那是因为事实确实如此！）

在查找的过程中，每当我们比较枢轴值与要查找的值的大小时，不妨输出一个二进制位来表示我们所做的决定（约定用 0 表示小于，用 1 表示大于）。

实际操作更便于理解，下面就实际操作一遍，具体如图 2-1 所示。

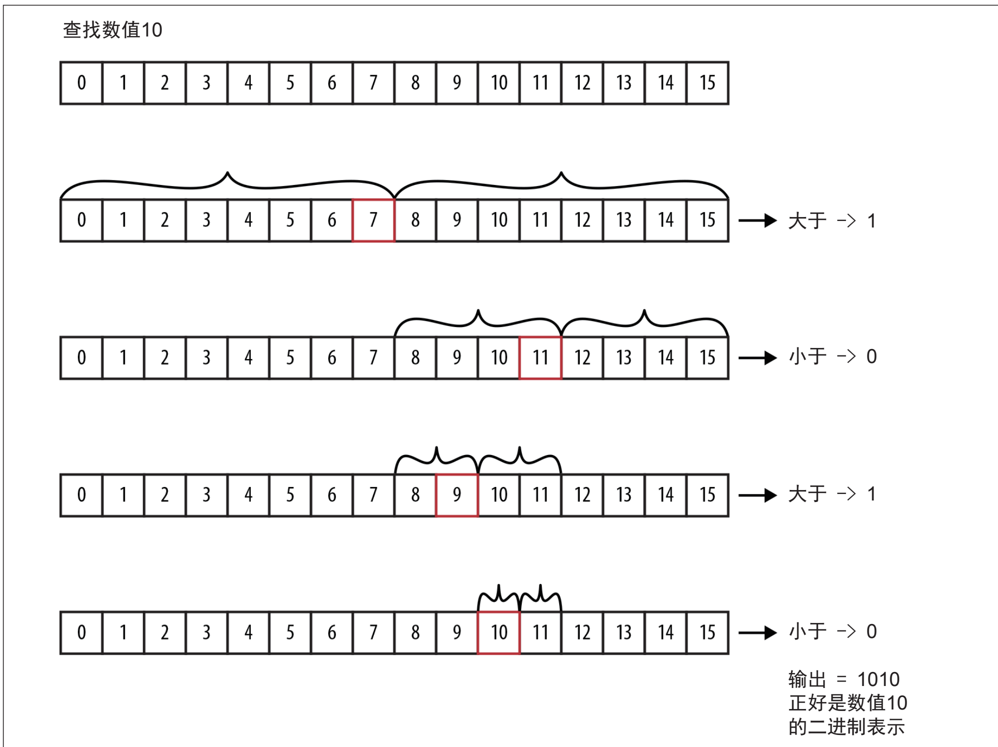


图 2-1：在一个有限的数值范围内进行二分查找。如果将每一步所做的决定（选择大值端还是小值端）都保存下来，最终就会得到要查找的数值的二进制形式

注 3：当然，如果一个数组包含偶数个元素，那么真正处于中间的元素是不存在的。这种情况下，可以根据喜好从中间偏左或中间偏右这两个元素中任选一个。

这里最终的输出值是二进制值 1010，这个值很有意思，它恰好是我们要查找的数值 10 的二进制形式。对元素个数等于 2^n-1 的数据集，我们通过记录为了明确描述一个数，做了多少次“是”或“否”的决定，来表示这个数。



如果你希望能在聚会中独处，不妨试试与其他人讨论这个话题。我敢肯定，这样做之后，你很快就会成为那个很尴尬的人。

2.2.2　熵：表示一个数所需要的最少二进制位数

可以看到，给定任意一个整数，我们都能将它转换为二进制形式。然而令人遗憾的是，给定一个整数，如果不经过二进制转换这一过程，我们很难直接知道它需要占几个二进制位。转换过程很无趣，但好在数学家已准备了下面的公式，让我们可以更轻松地处理这个问题[4]：

$$lb(x) = (log(x)/log(2)) = \text{表示一个数所需要的二进制位数}$$

从数学上来说，lb 将产生一个浮点数，例如，lb(10) = 3.321。

然而，从技术角度来说，由于现代的计算机硬件无法表示 3.321 个二进制位（这是因为二进制位已经是数据的最小单位，我们能使用的最小的二进制位数就是 1），因此我们必须对这个值向上取整，也就是使用向上取整函数，即 ceil（或 ceiling）函数，从而将上面的公式更新为下式（由于这是一个更整洁的版本，因此我们决定用大写字母来表示，以示区别）：

$$LOG2(x) = ceil(log(x)/log(2))$$

不过，这又产生了另外一个问题，因为从技术上来说，如果一个数正好是 2 的幂，那么通过这一公式所得的结果要比实际需要的二进制位数小 1。

下面以 2（或者任何其他 2 的幂）为例：

$$LOG2(2) = ceil(log(2)/log(2)) = 1$$

$$LOG2(4) = ceil(log(4)/log(2)) = 2$$

可以看到，从数学上来说，上面计算的结果都正确，但实际上在计算机系统中表示 2（二进制为 10）和 4（二进制为 100）分别需要 2 个和 3 个二进制位。因此，还需要对上述公式做一些修正，以确保在遇到 2 的幂这些特殊的数时所得的结果还是正确的。

$$LOG2(x) = ceil(log(x + 1)/log(2))$$

注 4：很抱歉又用到了数学，不过这是理解后面的内容所必需的。

为了让你更清楚地理解这一概念，我们来看下表，其中展示了关于 LOG2 的一些很有趣的数据，以及用二进制表示相应的数所需的二进制位的数量。

十进制数	LOG2（值）	二进制表示
0	1	0
1	1	1
2	2	10
3	2	11
4	3	100
7	3	111
15	4	1111
255	8	11111111
65535	16	1111111111111111
9.332622e + 157	525	……

因此，给定任意一个十进制整数，通过计算它对应的 LOG2 函数的值，我们就能知道用二进制来表示这个数最少需要多少二进制位。香农将一个变量对应的 LOG2 函数的值定义为它的**熵**（entropy），也就是用二进制来表示这个数所需的最少二进制位数。

2.2.3 标准的数字长度

数值的 LOG2 表示形式虽然高效，但对于制造计算机元件的方式来说并不实用。

这其中的问题在于，如果用最少的二进制位数来表示一个数，在解码相应的二进制字符串时会产生混乱（因为我们并不知道该数对应的 LOG2 长度），会与硬件的执行性能相冲突，两者不能兼顾。

现代计算机采用了折中的方案，用固定长度的二进制位数来表示大小不同的整数。最基本的存储单元是一个字节，由 8 个二进制位组成。在现代编程语言中，通常可用的整数的存储类型包括：短整型 16 个二进制位、整型 32 个二进制位、长整型 64 个二进制位。因此，对于十进制数 10，虽然其对应的二进制数为 1010，但在实际存储中是短整型，在计算机中的实际表示为 0000000000001010。显然，这样做浪费了很多的二进制位。

这里需要指出的是，在现代应用程序开发中，我们所用到的绝大多数算法使用预先设定好的固定的二进制位长度，而不是通过 LOG2 函数计算出的二进制位长度。这就是信息论与实际实现层面的差别。在计算机内存中，我们遇到的任何二进制位流都会舍入到下一个字节对齐的大小。这可能会让人困惑，比如当我们只存储了 7 个二进制位的数据时，计算机仍然会报告说我们所存储的数据长度与计算机一次读取的最小字节数相同。

实际中的数据压缩目标，是尽可能接近理论上的压缩极限。这就是为了更好地学习和理解压缩算法，从而顺利掌握本书后面的内容，我们只从 LOG2 的角度去考虑问题的原因。

第3章

突破熵

3.1 理解熵

由于没有更好的方法，因此香农博士将一个数对应的 LOG2 函数值称为该数的**熵**，也就是表示这个数所需要的最少二进制位数。他进一步将熵的概念（既然已经提出了这一术语了，为什么不重复利用呢……）扩展到整个数据集，也就是表示整个数据集所需要的最少二进制位数。他完成了所有这方面的数学工作，并给出了下面这个优美的公式来计算一个集合的熵 $H(S)$[1]：

$$H(S) = -\sum_{i=1}^{n} p_i \mathrm{lb}(p_i)$$

这个公式乍看起来可能有些吓人[2]，因此我们将它拆开来分析[3]。

> **熵（名词）**
>
> 一个热力学量，表示的是一个系统中无法转换为机械功的热能的量，通常被解释为该系统的无序度或随机度（物理学中的解释）。
>
> 无序或缺乏可预测性，逐渐退化为混乱（H. P. Lovecraft）。
>
> **对在特定的消息或语言中信息传输速度的一种对数度量（信息论中的解释）。**

注 1：香农决定借用玻尔兹曼 H 定理中的符号 H（即大写的希腊字母 eta）来表示熵。

注 2：注意，熵的计算公式中使用的 lb() 是数学意义上的，与我们在第 2 章定义的 LOG2() 函数不同。因此，这里通过 lb() 计算出的结果不会向上取整，同时也允许该值为负。

注 3：你可以在 Rosetta Code 网站上找到这一算法在各种语言中的实现。

更实际具体一些，我们先来看一组字母，例如：

$$G = [A,B,B,C,C,C,D,D,D,D]^{4}$$

首先，计算这个数据分组 G 中所包含的元素集合 S（这里所说的“集合”是数学意义上的集合，即集合中的每个元素只出现一次，且元素之间的顺序无关紧要）。

$$S = \text{set}(G) = [A,B,C,D]$$

这就是 G 中所包含的不同的符号的集合。

下一步，计算集合中的每个符号在数据分组中出现的概率，其计算公式为

$$P(v_i) = \text{count}(v_i)/\text{len}(G)$$

这个计算公式是说，一个符号 v 出现的频率或概率 P，等于这个符号在数据分组 G 中出现的次数（也就是公式中的 count(v)）除以数据分组 G 的长度。

为了将数学转化为图表，我们来计算 G 中每一个符号出现的概率。因为 G 中共有 10 个符号，所以 len(G) 等于 10，因此每个符号的概率都等于 0.1 的倍数。

符号	次数	概率
A	1	0.1
B	2	0.2
C	3	0.3
D	4	0.4
	和必然等于：	1.0

既然已经计算出了每一个符号出现的概率，下面就可以进一步计算香农所定义的数据分组 G 的熵 H。现在再回过头来看上面那个优美的公式，相信你已经不觉得它吓人了，因为它要比你想的简单得多：

$$H(S) = -\sum_{i=1}^{n} p_i \,\text{lb}(p_i)$$

第一步，对于每个符号，将其出现的概率乘以此概率以 2 为底的对数；然后，将第一步所得的数相加求和，再取其相反数，这样就得到了这一数据分组的熵。

下面来计算前面所举的示例 G 的熵，如下表所示。

下标 i	概率 p_i	lb(p_i)	$p_i \times$lb(p_i)
1	0.1	–3.321	–0.3321
2	0.2	–2.321	–0.4642
3	0.3	–1.736	–0.5208
4	0.4	–1.321	–0.5284
		总和	–1.8455

注 4：其中的字母是否排好序无关紧要，它不会对熵产生影响，本章后面会提到这一点。我们之所以选择排好序的示例，是为了让读者一眼就能看出每个字母的个数。

对最后一列求和，得到 –1.8455（允许有些小误差）。求熵公式的最前面还有一个负号（即求和符号∑前面的那个负号），因此得出结论：表示这组数据每个符号平均约需要 1.8455 个二进制位。搞定！

3.2 熵有什么用处呢

因为 G = [A,B,B,C,C,C,D,D,D,D] 的熵 $H(G) \approx 1.8455$，所以我们可以大致认为平均每个值用 2 个二进制位（通过向上取整运算获得）就可以对 G 进行编码。

可以像下面这样赋给每个符号 2 个二进制位的编码值：

A -> 00
B -> 01
C -> 10
D -> 11

这样一来，用二进制编码表示的 G^e 就会是下面这样：

$$e = [00,01,01,10,10,10,11,11,11,11]$$

这样编码之后，得到的 G^e 大小就是 20 个二进制位（在大多数教科书中表示为 $|G^e|$）。

下面是很有趣的部分：为了得出 G^e 的最终大小，实际上不需要进行编码这一步，只需要将熵 H 的值向上取整[5] 再乘以 G 的长度（即 $|G|$）就能得出结果：

$$H(G) \times |G| = 2 \times 10 = 20 \text{ 个二进制位} = |G^e|$$

根据香农的熵的定义，这就是表示这一数据集所需要的最小二进制位数[6]。

因此，总结起来就是，为了使表示某个数据集所需的二进制位数最少，数据集中的每个符号平均所需的最小二进制位数就是熵。

3.3 理解概率

从本质上来说，香农所定义的熵，是以一种倒排序的方式建立在数据流中每个符号出现概率的估算之上的。

总的来说，一个符号出现得越频繁，它对整个数据集包含的信息内容的贡献就会越少，这看起来似乎完全违背直觉。

注 5：记住实际上二进制位数根本不可能是小数。
注 6：这样的说法是错误的，稍后会说明为什么。

钓鱼就是这样一个真实的例子。设想你坐在岸边草坪中的躺椅上，戴着漂亮的帽子，手拿卷筒和连杆，望着河面和鱼浮。每隔几分钟，你就会看一眼鱼浮，发现它并没有变化，但每隔一小时左右，就会有鱼咬钩，这才是你真正感兴趣的事情。也就是说，很长的时间里没有什么有用的信息，真正有用的信息偶尔才会出现。如果用 0 表示没有鱼，用 1 表示鱼上钩，那么你记录的信息里很容易出现这样的片段：00000000100000000010000000000001。

从统计上来说（从运动角度来说同样如此），我们感兴趣的是鱼在咬钩这样的事件，其他的都是冗余信息。

比喻就说到这里，下面来看一些数值的例子。下表展示了一些概率的集合（这里我们不关心实际的符号）以及这些集合相应的熵。

概率集合 $P(G)$	熵 $H(G)$	对于一个包含1000个符号的数据集来说……
[0.001, 0.002, 0.003, 0.994]	0.06	其中会有 994 个相同的符号
[0.25, 0.25, 0.003, 0.497]	1.53	其中会有 497 个相同的符号
[0.1, 0.1, 0.4, 0.4]	1.72	其中会有 2 个符号，各出现 400 次
[0.1, 0.2, 0.3, 0.4]	1.84	其中会有 1 个符号是主要符号，但不占大多数
[0.25, 0.25, 0.25, 0.25]	2	共有 4 个符号，各出现 250 次

那么，应该如何理解这张表呢？

第一行中，第四个符号出现的概率最大，可以说占据了绝大多数的出现机会。换句话说，这个数据集主要是由其中的一个符号组成的，偶尔会有其他符号随机出现。由于这个数据流中的绝大多数内容是其中一个符号，这就意味着这个数据集中包含的总体信息很少，因此对应的熵值也很小。

再来看表中的最后一行，可以看到 4 个符号等可能地出现，因此它们对整个数据流内容的贡献相同。结果是这个数据集包含了更多的信息，因此需要用更多的二进制位来表示它。

举个类似的例子，打地鼠游戏之所以很有趣，是因为地鼠出现在每个洞的可能性相同，所以事先永远不会知道地鼠会从哪个洞里钻出来，这就使它比钓鱼有趣得多。

3.4 突破熵

数据压缩领域的最前沿都是关于如何改变熵的。事实上，整个数据压缩科学界的人士都认为熵是互联网上的一大“谎言”。

真相是，实际上[7]，通过利用真实数据的两个性质，我们完全有可能将数据压缩得比熵定义的还要小。按照香农对熵的定义，他只考虑了符号出现的概率，完全没有考虑符号之间的排序。而对真实数据集来说，排序是一项基本的信息，符号之间的关系同样如此。

注 7：这句话可以从“理论”上来理解。

例如，排好序的 [1,2,3,4] 和没有排序的 [4,1,2,3] 这两个集合，按照香农的定义，两者的熵相同，但是凭直觉我们就能发现，其中的一个集合包含了额外的排序信息。我们再来看一个元素为字母的例子，[Q,U,A,R,K] 和 [K,R,U,Q,A] 这两个集合有相同的熵，但 [Q,U,A,R,K] 这个集合表示的是英语中一个有意义的单词，而且字母的出现也有一定的规则，比如字母 Q 后面通常会跟着字母 U。

下面举几个例子来看一下如何利用数据的性质来突破熵。（请撸起袖子，我们将要压缩一些数据！）

突破熵的关键在于，通过利用数据集的结构信息将其转换为一种新的表示形式，而这种新表示形式的熵比源信息的熵小。

3.4.1　示例1：增量编码

元素递增的集合 [0,1,2,3,4,5,6,7] 称为集合 A。

现在，打乱集合 A 中元素的顺序，得到集合 $B = [1,0,2,4,3,5,7,6]$。

根据信息论的观点，这两个集合具有如下独特的性质：

- 所有的符号都等可能地出现，并且没有重复的符号；
- 集合 A 和集合 B 的熵 H 相等，都等于 3。

因此，根据香农的定义，每个符号都需要用 3 个二进制位来编码，而每个集合则需要 24 个二进制位。然而最终的结果是，我们很容易就能突破熵的限制，用更少的二进制位对集合 A 进行编码，具体方法如下。

集合 A 实际上是数的一个线性递增序列。因此，无须对其中的每个数都进行编码，而是可以对整个数据流进行转换，将各个数编码为其与前一个数的差。所以，编码后的 A 会是这样：

$$[0,1,1,1,1,1,1,1]$$

这一数据流的熵 $H(A) = 1$，结果还不错吧？

这样的转换一般称为**增量编码**（delta coding），也就是将一系列的数转换为其与上一个数的差后再编码[8]。

下面来讨论集合 B。由于 B 中的数并不是线性递增的，因此增量编码的方法不会起作用，这样操作之后我们会得到集合 [1,−1,2,2,−1,2,2,−1]，其熵为 2，乍一看这个结果还不错。然而，如果真的这样做，那么首先需要用 16 个二进制位将多重集合 B[9] 编码为 [01,00,10,10,00,10,10,00]。此外，还需要告诉解码器编码“00”表示的是符号 –1，这就需

注 8：如果这里不太理解，不用担心，第 8 章会深入讨论这一编码方法。

注 9：多重集合是指同一个元素可以多次出现的集合。

要额外的空间来存储。因此，即使能节省空间，也节省不了多少。（实际上，对某些集合来说，与直接对数据进行编码相比，增量编码所需要的二进制位甚至要更多。）

> **顺序很重要**
>
> 根据熵的定义，符号之间的顺序无关紧要，但增量编码证明事实并非如此。如果相邻的值之间高度相关，那么用增量编码的方法可以转换数据，使其熵变得更小。

3.4.2 示例2：符号分组

假定你遇到了字符串 S = “TOBEORNOTTOBEORTOBEORNOT”，它包含了不同符号的集合 [O,T,B,E,R,N]，而熵 $H(S) = 2.38$。

任何人看一眼这个字符串，都能意识到其中含有重复的单词。因此，如果不是将单个的字母当成符号，而是把单词当成符号，情况又会如何呢？这样一来，我们有单词集合 [TO,BE,OR,NOT]，其熵 $H(S) = 1.98$。

因此，对这样的数据流来说，用单词作为符号，得到的熵值会更小。那么，采用这种符号分组的方法，我们能走多远呢？再观察这个字符串，我们发现“TOBEORNOT”这个词组出现过多次，那么能否将词组当成一个符号呢？

下面试着计算一下：

set(S) = [TOBEORNOT,TO,BE,OR]，其熵 $H(S) = 1.92$ [10]

显然这个熵值更小！[11]

> **字符分组很重要**
>
> 上述字符分组的例子表明，如果数据集中存在连续值组合出现多次的情况，就可以利用这种情况来减小熵。一般来说，通过最佳符号分组预处理数据，会得到一个较小的熵值。

3.4.3 示例3：排列

一件有趣的事是，示例 1 中的集合 B[1,0,2,4,3,5,7,6] 是集合 A[0,1,2,3,4,5,6,7] 打乱后的版本，或者说是 A 的一个排列。

注 10：当然，虽然从理论上来说这两种分组方法的结果不同，但实际上对这个小数据集来说，两种方法一样，即每个符号都需要 2 个二进制位，不过这里主要关心的是超越香农。

注 11：值得一提的是，关于符号分组，一般有最佳的分组方法，而且有一个专门的数据转换研究领域帮我们找出最佳的分组方法，通常被称为“字典编码器”（Dictionary Encoders），第 7 章会讨论这个话题。

在数学中，排列这一概念是指重新安排或者改变一个集合的所有元素的次序或者顺序。

实际上，一个排列就是原来的集合打乱顺序后的一个版本，这里，我们会关注集合元素之间的顺序，并且确保没有重复元素。从经典的定义来看，只有同一组数的不同顺序才算排列，比如可以说 [2,1,3,4] 是 [1,2,3,4] 的一个排列，但 [5,2,7,9] 就不是。

排列很难压缩是出了名的。（有些人甚至认为基本不可能，我们无法确定他们是否真的理解这个词的含义。）原因很简单，从熵的角度来看，一个排列是不可压缩的，因为排序本身并不包含什么信息（这是因为它已经不再是有序的）。由于每个值出现的可能性相同，因此需要相同数量的二进制位来表示。

集合 $Q = [2,1,3,0]$ 编码后的大小等于 $\text{len}(Q) \times \text{lb}(\max(Q)) = 8$ 个二进制位 [12]，可以将其大小的计算公式归纳为 $N \times \text{LOG2}(N)$，请记住这个公式。当对数据压缩、信息论和熵有更多的了解时，这个值会一再出现，提醒我们在这个宇宙中我们是多么微不足道。

通过消除编码法压缩排列

还记得前面说过排列不可压缩吗？不好意思，我们说谎了。不过这个谎不大，而且有必要撒这个谎，这样你才明白形势的严峻。同时，我们也要对你说一声“抱歉”。事实上，排列是可以稍微压缩的，但这个过程没什么意思，也没多少实际价值。不过，我们还是准备看一看。

我们来看集合 C=[5,7,1,4,6,3,2,0]。

根据集合的元素值对它进行编码，由于最大值为 7，因此每个元素都需要 3 个二进制位，编码后有：

101 111 001 100 110 011 010 000

一共是 24 个二进制位。

现在，换一种方法，通过索引来编码，具体步骤如下。（如果你喜欢传统的方法，不妨拿着纸和铅笔跟着做。）

第一轮操作

创建一个包含 8 个元素的空数组，每个下标为 i 的元素保存的值为 i，如下图所示。

0	1	2	3	4	5	6	7	**数值**
0	1	2	3	4	5	6	7	索引

(1) 从集合 C 中的第一个元素开始，其值为 5。

注 12：这里我们用 $\max(A)$ 来表示集合 A 中的最大元素，即按递增排序后集合 A 中的最后一个元素。

(2) 计算该数的空闲位置下标（Free-Slot-Index），即找出其值为 5 的元素的下标。在这个例子中，5 的下标就是 5。
(3) 计算出你需要多少二进制位才能对这一下标进行编码，这可以通过计算元素个数的 LOG2 得出。由于一共有 8 个元素，因此 LOG2(8)=3，即 3 个二进制位。所以，可以用 3 个二进制位对 5 进行编码，即 101。
(4) 将值为 5 的元素从数组中删除。

输出流为 101，新的工作数组如下图所示。

0	1	2	3	4	6	7
0	1	2	3	4	5	6

第二轮操作

(1) 从集合 C=[5,7,1,4,6,3,2,0] 中取第二个元素，即 7。
(2) 从数组中找出其值为 7 的元素下标。现在，7 的下标为 6。
(3) 数组中还有 7 个元素，而 LOG2(7)=3，因此用 3 个二进制位对下标 6 编码并输出，得到 110。
(4) 将值为 7 的元素从数组中删除。

输出流为 101 110，新的工作数组如下图所示。

0	1	2	3	4	6
0	1	2	3	4	5

第三轮操作

(1) 从集合 C 中取第三个元素，即 1。
(2) 从数组中找出其值为 1 的元素的下标。现在，1 的下标为 1。
(3) 数组中还有 6 个元素，而 LOG2(6)=3，因此用 3 个二进制位对下标 1 编码并输出，得到 001。
(4) 将值为 1 的元素从数组中删除。

输出流为 101 110 001，新的工作数组如下图所示。

0	2	3	4	6
0	1	2	3	4

第四轮操作

(1) 取集合 C 的第四个元素，即 4。
(2) 从数组中找出 4 对应的下标，即 3。
(3) 数组中还有 5 个元素，而 LOG2(5)=3，因此用 3 个二进制位对下标 3 编码并输出，得到 011。
(4) 将值为 4 的元素从数组中删除。

输出流为 101 110 001 011，新的工作数组如下图所示。

0	2	3	6
0	1	2	3

第五轮操作

现在，事情变得有趣起来。（我们已经从数组中取出了一半的元素，[~~5,7,1,4~~,6,3,2,0]。）

(1) 取第五个元素，即 6。
(2) 从数组中找出 6 对应的下标，即 3。
(3) 此时，数组中元素的个数变为 4，而 LOG2(4)=2, 因此只需要 2 个二进制位就可以对下标进行编码。
(4) 用 2 个二进制位对下标 3 编码并输出，得到 11。
(5) 将值为 6 的元素从数组中删除。

输出流为 101 110 001 011 11，数组如下图所示。

0	2	3
0	1	2

最后的操作

(1) 下一个值为 3。
(2) 其对应的下标为 2。
(3) 用 2 个二进制位对下标 2 编码并输出，得到 10。
(4) 将值为 3 的元素从数组中删除。

输出流为 101 110 001 011 11 10，数组如下图所示。

0	2
0	1

(1) 下一个值为 2，其对应的下标为 1，用 1 个二进制位进行编码。
(2) 最后一个值为 0，其对应的下标也为 0，同样用 1 个二进制位进行编码。

最终的输出流为 101 110 001 011 11 10 1 0，其长度为 18 个二进制位。

按上面的方法编码，最终节省了 6 个二进制位。下面通过下表来进行比较。

输出：

输入	5	7	1	4	6	3	2	0
下标	5	6	1	3	3	2	1	0
二进制位	101	110	001	011	11	10	1	0

对数直接进行编码时，共需要 24 个二进制位，而对下标编码时，只需要 18 个二进制位，也就是节省了大约 25% 的空间。好了，现在我们已经知道了这样做可以节省空间，但是……

为什么这样做能节省空间

我们知道，对于包含 *N* 个整数，取值范围为 0~*N*–1 不重复的全排列，一共有 *N*！（称为 *N* 的阶乘）种可能。因此，第一个值出现后，我们就知道它不会再次出现。也就是说，去掉第一个值后，就只剩下 (*N*–1)! 种可能；而去掉第二个值后，就只有 (*N*–2)! 种可能了，以此类推。在某个点时，LOG((*N*–*X*)!) 的下标值与 LOG(*N*!) 的下标值之差就能变成整数。因此，可以用更少的二进制位来确定还剩下哪些可能性。

无论排列的大小多大，这种方法都适用。通过这种方法来编码，总能确保最终所产生的数据流比通过熵计算的小。例如，如果你遇到的是包含从 0 到 65 535 这些整数的一个排列，不管这些整数的顺序如何混乱，你总可以将这些数据压缩到原空间的 90%。实际上，只节省这么少的空间通常不值得我们这样去做。

解码工作则以相反的方式进行。最开始时，我们有一个空的数组，然后从数据流中读出 LOG2（# 没有赋值的元素个数）个二进制位，这表示的是该元素的下标，通过下标我们就知道它原来所代表的值。具体操作如下。

(1) 一共有 8 个没有赋值的元素，因此从输入流中读出前 3 个二进制位，这里是 101。
(2) 二进制 101 对应的值为 5，下标 5 对应的值为 5，因此第一个数是 5。现在，将 5 从数组中删除。
(3) 还有 7 个没有值的元素，我们需要读取接下来的 LOG2(7)（即 3）个二进制位，其值为 110，对应的十进制下标为 6，因此第二个数是 7。
(4) 接下来，请按照上面的方法解码剩下的元素。

3.5 信息论与数据压缩

上面这些简单的例子说明，在谈到信息论和熵时，还是有一些回旋的空间的。需要记住的是，熵定义的只是在对数据流进行编码时，每个符号**平均**所需的最小二进制位数。这就意味着，有些符号需要的二进制位数比熵小，而有些符号需要的二进制位数则比熵大。

数据压缩算法的**艺术**，就在于真正试图去突破熵的限定，或者说是将数据转换成一种熵值更小的、新的表现形式。这可以说是真正的舞蹈：信息论已经制定了相应的规则，数据压缩则以斗牛士的热情巧妙地避开了这些规则。

亲爱的读者，事实就是这样，这也是这本书的全部内容：怎样应用数据转换以创造熵更小的数据流（然后再用适当的方法进行压缩）。理解信息论与数据压缩之间的相互作用，将有助于你在当今这个信息世界更全面地看待这两者之间的相互协调与让步。

柯尔莫哥洛夫复杂性

正如前面所讨论的那样，在对压缩进行评价时熵不是一个好指标。[13]

其实，还存在其他更准确的度量复杂性的方法，但这些方法在使用上还没有真正地标准化。

柯尔莫哥洛夫复杂性（Kolmogorov complexity），度量的是确定一个对象所需要的计算资源。它以数学家安德雷·柯尔莫哥洛夫（Andrey Kolmogorov）的名字命名，以纪念他在 1963 年发表了这方面的第一篇论文。

举个例子，考虑以下两个由 32 个小写字母和数字组成的字符串：

abababababababababababababababab
4c1j5b2p0cv4w1x8rx2y39umgw5q85s7

可以写一个简单的 Python 程序来生成第一个字符串：

```
< v = 'ab' * 16 >
```

注意，这个生成字符串的程序要比字符串本身小。作为一种有效的数据压缩方法，你完全可以把这段程序发送给某个人，然后让他运行程序重新生成源字符串。

而第二个字符串没有什么规律，生成它的程序要比源字符串大。因此，这样做谈不上压缩。

下面做一个简单的总结。

熵，指为了唯一地描述一段数据所需要做出的“是”“否”回答的次数。

柯尔莫哥洛夫复杂性，指为了准确地生成数据，所需要的生成程序的大小。

可以证明，任何字符串的柯尔莫哥洛夫复杂性顶多比字符串本身的长度大几个字节（基本上，也就是一个程序输出字符串的每个元素）。那些柯尔莫哥洛夫复杂性要比字符串本身小很多，就像上面的所举的 abab 的例子，可以认为这样的字符串都很简单。

当开始讨论用**逻辑综合**（logic synthesis）或者**程序综合**（program synthesis）进行数据压缩的时候，柯尔莫哥洛夫复杂性就开始真正起作用了，本质上它取的是数据集以及反向生成产生字符串的程序的二进制位流。

平心而论，这样的解释有些大而化之。熵远远算不上最佳解决方法，但它的确是很多人信任的“足够好的”度量标准。要找到一个更准确的解决方法，涉及对数据信息和分析空间的大量随机探索。这里，需要着重说明的是，虽然已有将近 50 年的历史，但是数据压缩作为一个学科仍然很年轻。不是所有的问题都有答案，而这才是你真正需要去努力探索的。

注 13：平心而论，只考虑统计压缩时（见第 4 章），熵还不失为一个好的标准，但是在接触上下文压缩方法之后，你肯定想把它扔出去。

第 4 章
VLC

上一章所举的例子说明了两个问题：一是可以通过用更少或更多的二进制位对某些符号编码来节省所需要的总空间；二是当数据集中有重复符号时，这个方法就不太有用了。我们必须面对这个问题，因为在真实的数据集中，符号重复几乎无法避免。

这就是为什么 LOG2 方法无法正确地表示一个数据集中所真正包含的信息内容。本章将向你展示怎样利用概率和重复来做一些漂亮的工作，最终产生让人印象深刻的压缩结果。

4.1 摩尔斯码

在讨论真实数据的传输之前，让我们先回到电报和摩尔斯码的时代。

从 1836 年着手，3 位美国人——画家 Samuel F. B. Morse、物理学家 Joseph Henry 和机械师 Alfred Vail——共同发明了第一套电报系统。这套系统通过电线来发送电流脉冲[1]，这些脉冲与位于电报系统接收端的电磁体进行交互，产生了可以听得见的声音，或者人们在发声装置下放一条以固定速度运行的纸带，纸带就能将收到的信号记录下来。

电报是一项了不起的发明，因为它可以远距离地传递人类信息。慢慢地，电线消失了（见图 4-1），它最终演变成了人们口袋中的移动设备。

注 1：可以听一下这个示例，https://www.youtube.com/watch?v = xsDk5_bktFo。

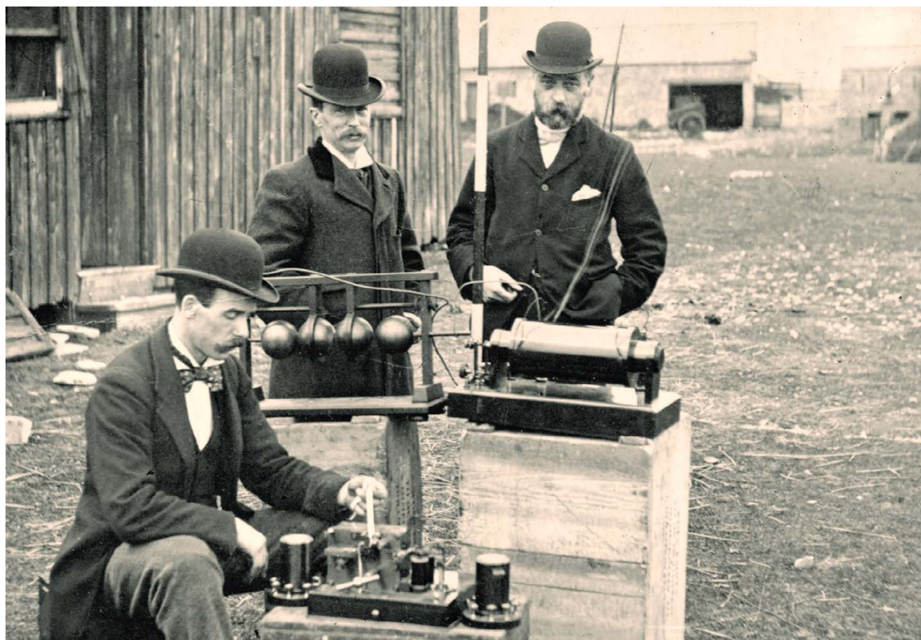


图 4-1：1897 年 5 月 13 日，在弗拉特霍姆岛的一次演示中，英国邮政工程师们在检查 Guglielmo Marconi 的无线电报设备。这是世界上第一次跨海无线电传输演示，从莱弗诺克角到弗拉特霍姆岛，距离大约为 4.8 千米。图片来源：维基百科

有了设备后，发明者遇到的问题就是如何表示人类的想法，并且这种表示方法是电流信号能传输的，比如说用语言。对操作人员来说，这个设备本身的操作方法很简单：按下电报按钮就能进行连接并通过电线传输电流；松开按钮，电流传输就中断。即使早在 19 世纪二进制编码还没有发明时，这套系统已经在应用同样的思想来传递信息了。

或许，最简单的编码文本信息的方法，就是用数字 1~26 来编码 A~Z 的英文字母。这样，我们就能通过脉冲的次数和组合来确定传送的是哪个数。例如，可以将“THE HAT”翻译为 20 − 8 − 5 8 − 1 − 20。实际上，要想使系统能真正工作，还要有方法来区分单词、空格和标点符号，当然还有结束符（end-of-message），但通过对单词进行编码，我们已经抓住了问题的实质。

不过，有一点要记住，那就是所有这些信号的传输都需要人不停地按电报设备的按钮。因此，发送“THE HAT”与发送“FAT CAT”或“TIP TOP”所需要的人工操作次数相同。如果一个邮政局每天要发 100~200 封平均 50 个词的电报，这件事情就会令人抓狂。显然，这是因为发送一次信息所需要的人工操作次数太多。物理硬件（发报机设备）和人工硬件（也就是操作人员的手腕）的磨损比预期的要快，解决方法则是使用统计来减少工作量。

我们都知道，在英语中有一些字母比另外一些字母使用得更频繁，比如字母 E 会在 12% 的时间里用到，而字母 G 则只在 2% 的时间里用到。如果操作人员每天发送的字母“E”更多，那么是不是应该让这样的操作变得更快、更简单呢？

最终，摩尔斯码被发明出来。

摩尔斯码为英语字母表中的每一个字符都分配了或长或短的脉冲，一个字母用得越频繁，其编码也就越短、越简单。因此，英语中最常用的字母“E”的编码最短，用一个点表示；而字母“X”的编码毫无疑问则很长；所有的数字都用 5 个脉冲表示。图 4-2 显示了摩尔斯码的原始字符集。

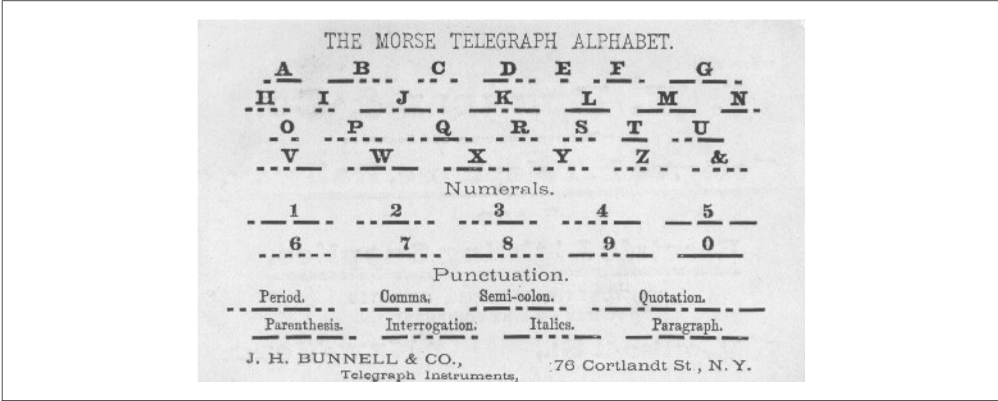


图 4-2：摩尔斯码根据各个符号在英语中出现的概率来为其分配点和划。一个符号出现得越频繁，其对应的编码就越短。这张图是摩尔斯码的一个早期版本，由电报公司专门为传输较小的信息集而设计。从那时起，摩尔斯码就一直在演变，现在的摩尔斯码看上去大不相同了

即使是追溯到 19 世纪，这也是对符号分配变长编码（variable-length codes，VLC）的最初实现之一，其目的则在于减少传输信息过程中所需要的总工作量。

有理由相信，在早期对信息论的研究中，克劳德·香农（他是摩尔斯码方面的专家）正是利用了这一概念，由此创造了一个新的技术领域“数据压缩”的第一代技术，这些都是在 VLC 的启发下产生的。

4.2 概率、熵与码字长度

为了进行数据压缩，我们的目的很简单：给定一个数据集中的符号，将最短的编码分配给最可能出现的符号。

不过我们还是暂时先退回来，看一看熵是如何与此相关的。假定一个大的数据集中只有 [A,B] 两个符号，我们来计算一下每个符号出现的概率（也就是 P(A) 和 P(B)），并看一看

概率如何影响数据集的熵。下表展示了不同的样本概率与对应的数据集的熵。

P(A)	P(B)	数据集的熵H
0.99	0.01	0.08
0.9	0.1	0.47
0.8	0.2	0.72
0.7	0.3	0.88
0.6	0.4	0.97
0.5	0.5	1
0.4	0.6	0.97
0.3	0.7	0.88
0.2	0.8	0.72
0.1	0.9	0.47
0.01	0.99	0.08

观察这张表，可以马上发现以下这些规律。

- 当 P(A) == P(B)，也就是两个符号等可能出现时，数据集对应的熵取最大值 LOG2（符号的个数），此时数据集很难压缩。
- 其中一个符号出现的可能越大，数据集的熵值就越小，此时数据集也越容易压缩。
- 对只包含两个符号的数据集来说，两个符号互换概率不影响其熵值，我们可以看到 H([0.9,0.1]) 等于 H([0.1,0.9])。

这给我们为给定的符号分配 VLC，提供了以下重要的启示。

首先，随着数据集的冗余度下降，它的熵在变大，其最大值为数据集中不同符号个数的 LOG2 值。

例如，如下表所示，数据集中有 4 个以相同概率出现的符号，其熵为 $-4(0.25\mathrm{lb}(0.25)) = 2$。也就是说，为了对每一个符号进行编码，平均来说至少需要 2 个二进制位。

符号	概率	码字长度	码字
A	0.25	2 个二进制位	00
B	0.25	2 个二进制位	01
C	0.25	2 个二进制位	10
D	0.25	2 个二进制位	11

总结起来就是，如果一个数据集中包含 4 个等概率出现的符号，那么其熵为 2，对其中的每个符号进行编码都需要 2 个二进制位。

其次，数据集中一个符号出现的概率越大，整个数据集的熵就越小，数据集也就越容易压缩。

与摩尔斯码类似，我们可以通过使用可变字长的码字来改善这种情况，将最短的码字分配给最可能出现的符号。因此，作为对比，我们来考虑下表中包含 4 个符号且符号呈偏态分布的数据集，通过使用 VLC，其熵仅为 1.57。

符号	概率	码字长度	码字[a]
A	0.49	1 个二进制位	0
B	0.25	2 个二进制位	10
C	0.25	3 个二进制位	110
D	0.01	3 个二进制位	111

a 这张表看起来很奇怪，不是吗？为什么不能用码字 [0,1,00,10] 来对 [A,B,C,D] 进行编码呢？这是因为“前缀性质”，这是解码的需要，同时也是构造编码时的一个基本要求。不要着急，稍后会讨论这一性质。

给定字符串 AAABBCCD（可将其看成一个更大的数据集的子串），假定每个符号出现的概率相同，那么一共需要 16 个二进制位；如果符号出现的概率是如上表所示的偏态分布，那么只需要 13 个二进制位就可以完成编码工作。本例中节省的二进制位数虽然有限，但如果是有着成千上万个符号的真实数据集，节省的空间就相当可观了。

通过上表计算出来的熵，同样也支持更小的压缩后大小。熵约为 1.57，也就意味着，平均来说，表示每个符号所需要的最少二进制位数为 1.57。

既然实际上数据流中的二进制位数不可能是小数，那么得出的这个 1.57 又有什么意义呢？需要注意的是，这是因为我们所举的例子都很短，包含的总字符个数很少。为了得到与理论计算相接近的结果，输入流中必须要包含上千个符号。

最后，码字的长度与符号的出现概率密切相关，而与符号本身没有太大关系。

我们拿前面所举的例子来说明这个问题。如果只是将字母 A 和字母 D 的出现概率互换，那么码字的总长度不会改变，只是字母 A 和字母 D 的码字长度会互换，如下表所示。

符号	概率	码字长度
A	0.01	3 个二进制位
B	0.25	2 个二进制位
C	0.25	3 个二进制位
D	0.49	1 个二进制位

4.3 VLC

因此，对于给定的输入数据集，可先计算涉及的符号的出现概率，然后就可以通过 VLC 将字长最短的码字分配给最可能出现的符号，从而实现压缩数据。这真不错！

当然，这里还存在两大问题没有解决。第一，怎样在应用程序中运用 VLC 来进行压缩呢？第二，对于给定的数据集，怎样构造 VLC 呢？

4.3.1 运用VLC

一般来说，对数据进行 VLC 通常有 3 个步骤。

(1) 遍历数据集中的所有符号并计算每个符号的出现概率。
(2) 根据概率为每个符号分配码字，一个符号出现的概率越大，所分配的码字就越短。
(3) 再次遍历数据集，对每一个符号进行编码，并将对应的码字输出到压缩后的数据流中。

下面来对每一个步骤进行一些讲解。

1. 计算符号的概率

这个步骤包括画出数据集中所有符号的直方图。也就是说，我们需要遍历数据集，并计算出每个符号出现的次数，最终得到如图 4-3 所示的直方图，该图描述了每个符号与其出现次数（或者说频数）之间的对应关系。



符号	频数
T	5
O	8
B	3
E	3
R	3
N	2

图 4-3：一个示例字符串与其对应的直方图，该直方图描述了每个字符及其出现次数

2. 为字符分配码字

接下来，根据出现的频数对直方图进行排序（见图 4-4），然后给每个符号分配一个 VLC，从 VLC 集中码字最短的开始。其结果就是，一个字符出现得越频繁，分配给它的码字就越短，如此就实现了数据压缩。

3. 编码

用 VLC 的方法对数据流中的字符进行编码非常简单。只需要从数据流中一一读出各个符号，然后从码字表中查找出当前符号所对应的码字，添加到输出流中，再将所有的码字连接起来，就能得到最终的输出流。

处理完整个输入流之后，再将符号码字对应表添加到输出流的前面，如图 4-5 所示，这样解码的时候就能将输出流恢复为源数据。



TOBEORNOTTOBEORTOBEORNOT

符号	频数	码字
O	8	11
T	5	00
B	3	011
E	3	101
R	3	0100
N	2	0101

图 4-4：按照字符出现的频数对直方图进行排序之后，我们就能为字符分配码字。这张表通常也被称为“码字表”（codeword table）。注意，这里的码字并不表示标准的二进制数，这是编码和解码的需要（本章稍后会讨论这一内容）

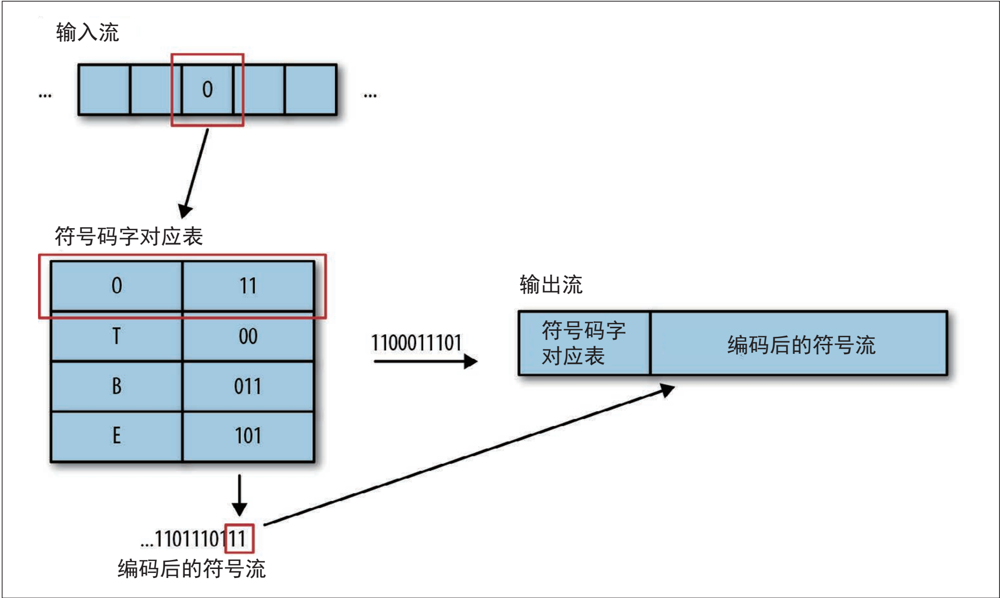


图 4-5：该图展示的是一个编码示例，从输入流中读取一个符号，然后通过符号码字对应表查出相应的码字，再将码字添加到输出流中。所有的符号完成编码后，还要将符号码字对应表添加到输出流的前面，以方便后面的解码

举个例子，要使用刚创建的符号码字对应表对“TOBEORNOT”进行编码，第一个字符是“T”，因此将相应的码字 00 添加到输出流，接下来是字符“O”，再将 11 添加到输出流。如此反复，最终得到“TOBEORNOT”所对应的输出流为 001101110111010001011100，共 24 个二进制位。

与 72 个二进制位的源数据流相比（即使平均每个字符只用 3 个二进制位），通过压缩节省了大约 11% 的空间。

4. 解码

一般来说，解码是编码的逆过程。如图 4-6 所示，解码时从输入流中读取二进制位，再通过符号码字对应表找出码字对应的符号，将此符号添加到输出流中。

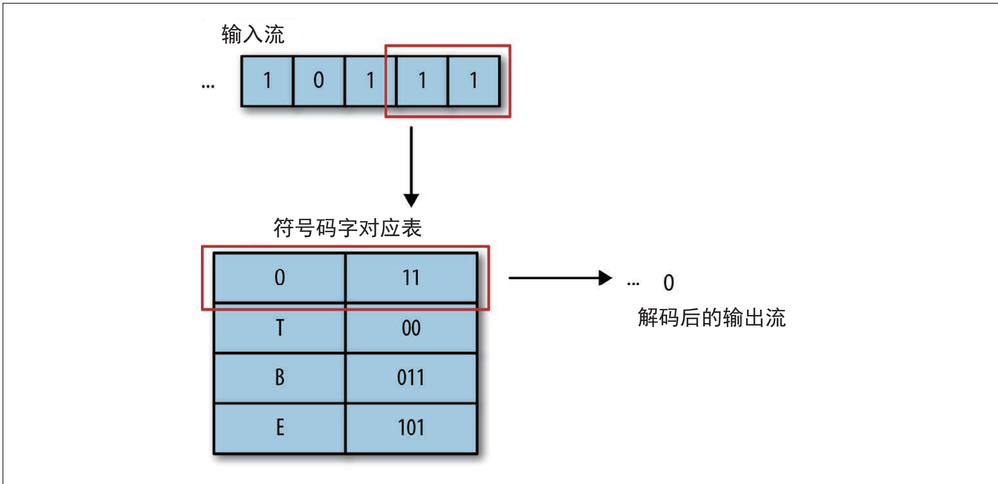


图 4-6：解码是编码的逆过程，从输入流中读取二进制位（也就是码字），再通过符号码字对应表找出码字对应的符号，然后输出这些符号

不过，由于码字的长度并非是固定的，因此解码过程还是稍微有些复杂。一般而言，解码的时候，我们会一二进制位一二进制位地读取数据，直到读取的二进制位流与其中的某个码字相匹配。一旦匹配，就会输出相应的符号，并继续读取下一个码字。

我们来看看上一个例子中编码后所形成的二进制位流 001101110111010001011100，并使用前述的符号码字对应表将它恢复为源字符串，步骤如下。

(1) 读取 0。
(2) 对照符号码字对应表，没有长为 1 个二进制位的码字。
(3) 继续读取，还是 0。对照符号码字对应表，00 是码字，输出对应的符号 T。
(4) 读取下一个二进制位，即 1。
(5) 继续读取，还是 1，在符号码字对应表找到码字 11，输出对应的符号 O。

现在，有意思的事情出现了。

(1) 读取 0，再读取下一个二进制位 1。此时，我们发现对应表中有多个匹配，下一个字符可能是 B、R 或 N。因此，需要继续读取第 3 个二进制位，得到“011”。现在，我们在表中找到唯一的一个匹配，字母 B。

(2) 按本书的惯例，剩下二进制位的解码由你来完成。当读到二进制位流的最后一位时，你应该已经恢复了源字符串（可以参考图 4-7）。

符号	码字
O	11
T	00
B	011
E	101
R	0100
N	0101

图 4-7：解码示例，从输入流中读取码字，再通过符号码字对应表找到相应的符号，最后将符号添加到输出流中

4.3.2 创建VLC

说到这里，我们想花一点时间指出摩尔斯码的一个细节。考虑用摩尔斯码“• • •”表示的消息，你能猜出它是什么意思吗？

我们看一下前面的表，3 个点表示的好像是字母“S”。

然而，再看一下这张表，它表示的也可能是 3 个“E”（即一个点表示一个符号）。

由操作员来决定某个电码到底表示什么字符，看起来有些让人怀疑。事实上，操作员大可以利用其他信息来决定某个电码到底表示什么字符。例如，在 19 世纪，“eee”几乎不可能表示有效信息，特别是在操作员知道具体语境的情况下（CATEEE 不是正确的英语单词，但 CATS 是）。

“手法”竞争

从技术上来说，现实中发报员可以通过用短暂的停顿来分割符号进行“欺骗”。不过停顿并非是统一的，而是取决于发报员的操作习惯。由于发报员的个人差异而造成的信号持续时间的差异，被称为发报员个人的“手法”(fist)。每个发报员的“手法”都不同。事实上，经验丰富的发报员仅凭手法就能认出某个发报员。甚至还有“好手法”和“坏手法”的概念，这和发报员发报时信号的清晰程度有关。

为了决定消息的内容，操作员需要做一些额外工作，而这项工作在数据压缩的世界中不能复制。这是因为我们面对的符号只有 0 和 1，而没有空白。正因为如此，将摩尔斯码作为码字使用，其结果并不理想。所以，我们需要找到一种能将 0 和 1 放在一起，并且解码器能生成唯一输出流的方法。

前缀性质

因此，在任意时刻，解码器都需要能确定到目前为止所读取的二进制位，是否与特定字符的码字相匹配，以及是否还需要继续读取下一个二进制位。为了确保正确，在设计 VLC 集的码字时，必须考虑两个原则：一是越频繁出现的符号，其对应的码字越短；二是码字需满足前缀性质。

我们先看看一个潜在的 VLC 是怎样崩溃的。假定我们有数据流 0101111，以及如下表所示的 VLC 对应表。

符号	码字
A	0
B	10
C	101
D	111

解码过程如下所示。

(1) 解码器读取 0，其对应的字符肯定是 A，因此将 A 输出。
(2) 解码器读取 1，它可能是字符 B、C 或 D 的码字的第一位。
(3) 解码器继续读取 0，现在可能字符的范围缩小为 B 或 C。
(4) 解码器继续读取 1，这里我们遇到了歧义：读取的二进制位 101，应该解释为 10 + 1（表示的是 B 与另一个字符的开始，该字符可以是 B、C 或者 D）**还是** 101（表示的是字符 C）呢？

此时，解码的过程已经进行不下去了，因为我们已不能确定读取的二进制位到底表示的是什么符号。如图 4-8 所示，我们设计的码字已经让我们无法区分不同的符号。

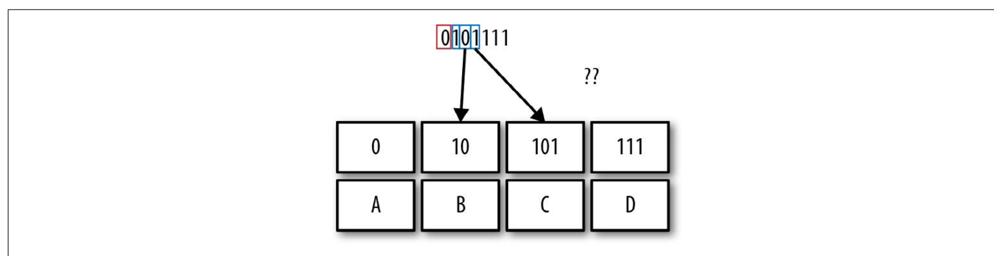


图 4-8：如果一个码字是另一个码字的前缀，那么用 VLC 解码二进制流就会很难

这里，我们提到了 VLC 的**前缀性质**，它的意思是：

在某个码字被分配给一个符号之后，其他的码字就不能用该码字作为前缀。

换句话说，每个符号都能通过其码字前缀唯一地确定，只有这样，VLC 才能正常工作。



前缀性质是 VLC 能正常工作所必须具有的性质。这也就意味着，与二进制表示相比，VLC 要更长一些。

折中后的结果是，我们得到的数据流会更长（因而也更大），但同时也能在事先不知道符号长度的情况下进行解码。由于那些常见的字符对应的码字很短，因此对那些少数几个字符出现频率特别高的数据流而言，VLC 的压缩率还是很高的。

4.3.3　几个VLC示例

本节将介绍一些曾被广泛使用并且现在依然发挥作用的 VLC。选择正确的 VLC 方法并赋给每个符号正确的码字，则是另外一项挑战，稍后会简要讨论这个问题。

这些 VLC 是怎样构造出来的

可以说这些编码的构造过程满含着血汗和眼泪。一位数学家板凳一坐十年冷，最终找到了一种将整数转换为 VLC 的独特方法。这种编码方法一般称为**通用编码**（universal codes），大致来说，就是为正整数赋上一个长度可变的二进制码字。通常来说，通用编码遵循以下原则：数值越小，其对应的码字也越短，因为假定小整数比大整数出现得更频繁。

这些 VLC 方法决定了用不同的方法分配码字长度。例如，有一种方法是数值每加 1 其码字的长度也加 1，所以 1 对应的码字长为 1 个二进制位，2 对应的码字长为 2 个二进制位，8 对应的码字长为 8 个二进制位，以此类推。也有相邻整数之间码字的二进制位数相差很大的，如 1 对应的码字长为 1 个二进制位，2 对应的码字长为 5 个二进制位，8 对应的码字长为 12 个二进制位。

这是因为每一个 VLC 方案都对符号的概率分布有特定的假定。如果第 N 个符号出现的概率是第 N+1 个的两倍多，那么其对应的码字应该要比第 N+1 个少 1 个二进制位。然而，如果第 N、N+1、N+2 和 N+3 个符号出现的概率基本相同，那么你可能会希望它们对应的码字长度相等。因此，接下来要介绍的 VLC 方案都与某个理想的概率分布相对应，对于那些与某个理想概率分布比较接近的数据集而言，选择对应的编码方案效果会更好。

所以，对于接下来要讨论的大多数 VLC 方案，我们会列出其相应的理想概率分布。不过要注意的是，虽然这很有趣但并非必须要知道，因此我们不打算详细解释。

值得指出的是，通用编码是一类特殊的前缀编码。还有其他类型的 VLC 方法，比如**唯一可译码**（uniquely decodable codes）与**非奇异码**（nonsingular codes），这里我们只是简单地告诉你这些编码方法的名称，不再展开。（本书中也不会使用这些术语来专门讨论这些编码方法，不过要注意的是，我们之所以提到这些，是因为每一种前缀编码都是唯一可译的和非奇异的。）

1. 二进制编码

第 2 章介绍了二进制编码，希望你还没有完全忘记。由于二进制编码在计算机中无处不在，因此它总是数据压缩的最终结果。

继 Peter Elias 之后，人们总是习惯用 B(n) 来表示整数 n 的标准二进制表示。这种表示通常被称为 beta 编码或**二进制编码**，不过它不满足前缀性质。例如，2 的二进制表示为 10，同时它也是 4 的二进制表示 100 的前缀。

给定 0~n 的任意整数，我们都能用 1 + floor(lb(n)) 个二进制位来表示。也就是说，只要提前知道 n 的值，我们就能通过固定长度表示法来避开前缀问题。换句话说，如果知道有多少个数需要表示，我们很容易算出一共需要多少二进制位。然而，如果不能提前知道数据集中有多少个不同的整数，就不能用固定长度表示法。

走近 Peter Elias

Peter Elias 教授于 1923 年 11 月 23 日生于美国新泽西州新不伦瑞克市。他的父亲是托马斯·A. 爱迪生（Thomas A. Edison）实验室里的一位工程师。

Peter Elias 先在斯沃斯莫尔学院上了两年学，之后于 1942 年转学到麻省理工学院。1944 年，一拿到商业与工程管理学士学位，他就应征加入美国海军，成为一名无线电技师。1946 年，他以电子设备技术员大副军衔退役。之后，他在哈佛大学取得了硕士（工程科学硕士）和博士学位。从 1953 年到 1991 年，他一直在麻省理工学院任教，在此期间，他获得了荣誉称号并成为了资深讲师。

在纠错码方面（不过本书并没有涉及这方面的内容），Elias 是一位真正的专家。1955 年，他就引入了**卷积码**（convolutional codes），作为**分组码**（block codes）的一种替代方法。此外，他还建立了**二进制删除信道**（binary erasure channel），并提出了用纠错码的**列表译码**（list decoding of error-correcting codes）来代替**唯一可译码**（unique decoding）。如果想在晚餐前深入地阅读，我们推荐你读一读他写的著作。（这些书很有趣。）

2. 一元码

任意正整数 n，它的一元码表示都是 $n-1$ 个 1 后面跟着 1 个 0，例如，4 的一元码表示为 1110。[2] 所以，很容易得出，整数 n 的一元码长度也是 n 个二进制位。

同样，也容易看出一元码满足前缀性质，因此可以将它作为 VLC 使用。

随着整数 n 加 1，其一元码码字的长度也线性地加 1，因此其码字的长度 L 总是等于 n。而在二进制编码中，码字的长度每增加 1 个二进制位，能表示的整数则呈指数级增长。（还记得前面所示的 2^1、2^2 等列吗？）

因此，将一元码应用在那些前一个符号的出现概率是后一个符号 2 倍的数据集上，效果最佳。（也就是说，A 的出现概率是 B 的两倍，而 B 的出现概率又是 C 的两倍。）或者，如果你喜欢数学，我们可以说，对那些包含 N 个整数的数据集来说，如果每个整数 n 的出现概率 $P(n)$ 服从指数分布 2^{-n}，即 1/2、1/4、1/8、1/16、1/32，其他以此类推，我们就可以使用一元码进行编码。

下面展示的是理想概率下一元码的示例。

数	编码	理想概率
0	•（无法表示）	
1	0	0.5
2	10	0.25
3	110	0.125
4	1110	0.0625

解一元码时，只需要从输入流中一直读取直到遇到分隔符 0，然后数一下 1 的个数再加上 1，最后将这个值输出就可以了。

3. Elias gamma 编码

Elias gamma 编码主要用于事先无法确定其上界的整数的编码，也就是说，不知道最大的整数会是多大。

注 2：值得注意的是，你可以对这一表示取反，使之变为 0001，这背后的思想则是其中有些位是作为值来使用的，有些位则被当作分隔符。

该方法最主要的思想是不再对整数直接编码，而是用其数量级作为前缀。这样一来，相应的码字就由两部分组成，即与此整数相当的 2 的次幂再加上余数，编码方法如下所示。

(1) 找出最大的整数 N，使其满足 $2^N < n < 2^{N+1}$，并且将 n 表示为 $n = 2^N + L$ 这样的形式（这里 $L = n - 2^N$）。
(2) 用一元码表示 N。
(3) 将 L 表示为长为 N 的二进制编码，并加在步骤 (2) 中得出的一元码之后。（这一步很重要，正是有了这样的对称性，后面才能顺利解码。）

举个例子，当 $n = 12$ 时，我们会进行如下的编码。

(1) 由于 $2^3 = 8$，而 $2^4 = 16$，同时 $2^3 < 12 < 2^4$，因此 N 的值等于 3。
(2) 根据前面的说明，可以算出 L 的值为 $12 - 8 = 4$。
(3) $N = 3$，其对应的一元码为 110。
(4) $L = 4$，其对应的长度为 3 的二进制码为 100。
(5) 将前两个步骤得出的编码连接，就得到了最终的输出 110100。

下面再对一个稍微大一点的数进行编码，比如说 42。

(1) 由于 $2^5 = 32$，而 $32 < 42 < 64$，因此 N 的值等于 5。
(2) L 的值则等于 $42 - 32 = 10$。
(3) $N = 5$，其对应的一元码为 11110。
(4) $L = 10$，其对应的长度为 5 的二进制码为 01010。
(5) 将 11110 和 01010 连接，得到最终的输出 1111001010。

与简单的一元编码类似，Elias gamma 编码对那些整数 n 的出现概率 $P(n) = 1/(2n^2)$ 的情形比较适用。

下表展示了一些 Elias gamma 编码的例子（注意，编码中的 L 部分用斜体表示）。

n	$2^N + L$	编码
1	$2^0 + 0$	•（无法表示）
2	$2^1 + 0$	0***0***
8	$2^3 + 0$	110***000***
12	$2^3 + 4$	110***100***
42	$2^5 + 10$	11110***01010***

解 Elias gamma 码字时，方法也很简单。

(1) 以 12 为例，其对应的码字为 110***100***。
(2) 读取输入流直到遇到分隔符 0，因此读取的值为 1、1、0，可以计算出 $N = 3$。
(3) 读取剩下的 3 个二进制位 110，并将其从二进制转换为十进制，由此得出 $L = 4$。

(4) 将两部分相加，即 $2^N + L = 2^3 + 4 = 12$。

4. Elias delta 编码

Elias delta 编码则是另外一个变种。在这一方法中，Elias 还在前面加上了二进制的长度，使这一编码稍微复杂一些。

它的工作原理如下面的步骤所示。

(1) 将要编码的数 n 用二进制表示。
(2) 数一下 n 的二进制位数，并将这个位数转化为二进制，作为 C。
(3) 去掉 n 的二进制表示的最左边一位，这个值肯定是 1，可以推断出来。
(4) 将 C 的二进制表示加在去掉最左边一位后的 n 的二进制表示前。
(5) 在上一步骤所得的结果前加上 C 的二进制位数减 1 个 0 作为最终的编码。

下面还以 12 为例，对它进行编码。

(1) 将 $n = 12$ 表示为二进制 1100。
(2)12 的二进制表示共有 4 位，将 4 表示为二进制，即 C = 100。
(3) 去掉 n 的二进制表示的最左一位，得到 100。
(4) 将 C = 100 加到上一步骤所得的结果之前，得到 100100。
(5) 将 C 的二进制位数减 1，即 3−1 = 2，在上一步骤所得的结果前加上 2 个 0，由此得到 12 的最终编码为 00100100。

对那些整数 n 的出现概率 $P(n)$ 等于 $1/[2n(\mathrm{lb}(2n))^2]$ 的数据集来说，Elias delta 编码是理想的选择。

下表展示了一些 Elias delta 编码的例子。

十进制数n	$2^N + L$	编码
1	$2^0 + 0$	1
2	$2^1 + 0$	0100
8	$2^3 + 0$	00100000
12	$2^3 + 4$	00100100
18	$2^4 + 2$	001010010
314	$2^8 + 58$	000100100111010

要解码 Elias delta 编码得出的编码，可以按照下面的步骤进行。

(1) 读取并数 0 的个数，直到遇到 1。
(2) 将 0 的个数加 1，由此得到 C 的长度。
(3) 继续读取 C 的长度个二进制位，由此得到 C。
(4) 继续读取 C−1 个二进制位，在前面加上 1 并将其转为十进制数。

以前面得到的 00100100 为例进行解码。

(1) 读取并数 0 的个数，直到遇到 1；这里有 2 个 0。
(2) 加 1，得到 C 的长度为 3。
(3) 继续读取 3 个二进制位，得到 C = 100 = 4。
(4) 继续读取 4−1 即 3 个二进制位，即 100，在其前面加上 1，得到 1100，因此解码的结果为 12。

如果你不相信这一方法可以工作，不妨用数 314 为例先编码再解码，进行验证。

5. 其他编码方法

相信通过前面的介绍你已经掌握了这几种简单的 VLC 算法。不过必须要指出的是，在过去的 40 多年中，人们创造了数百种 VLC 算法，由于本书篇幅限制，我们无法一一进行介绍。

谷歌的 Varint 算法

VLC 主要存在以下几个问题，因而只能用于表示压缩数据流，没有其他应用。

- 它们不按字节 / 字 / 整型对齐。
- 对于大的数值 n,为了方便解码,其码字长度的增长速度一般会超过 $\mathrm{lb}(n)$ 个二进制位。
- 解码的速度很慢（每次只能读取一个二进制位）。

对那些需要处理很多大整数的系统来说，这些问题使得 VLC 无法应用。然而，在 21 世纪初，出现了一个新的可变长度整数模型，在搜索引擎和其他海量数据系统中得到了广泛应用。虽然它是以谷歌的 Varint 算法而流行的，但其中最基本的概念早在 1972 年就提出来了[4]，并在 2010 年作为“避免压缩整数”（escaping for compressed integers）而被重新引入。

Varint 是一种表示整数的方法，它用一个或多个字节来表示一个整数，数值越小用的字节数越少，数值越大用的字节数越多。

该方法将几个字节连接起来表示整数，并用每个字节的 MSB 作为布尔标志，来判断当前的字节是否为该整数的最后一个字节；而每个字节的低 7 位则用来存储该数的**二进制补码**（two’s complement representation）。下面就来看一些例子。

整数 1 可以用一个字节表示，因此它的 MSB 不需要设置，可表示为 00000001。

再来看整数 300，它的表示则要复杂一些：10101100 00000010。那么如何判断它表示的是 300 呢？

首先，需要删掉每个字节的 MSB，因为它的作用只是判断当前字节是否是最后一个字节（正如你看到的那样，第一个字节的 MSB 已经设置为 1，因为用 Varint 方法来表示，该数需要多个字节）。

注 3：参见 L. Thiel 和 H. Heaps 的论文 *Program Design for Retrospective Searches on Large Data Bases*，载于 *Information Storage and Retrieval*，1972 年第 8 卷，第 1 期，第 1~20 页。

10101100 00000010 → 0101100 0000010。

接着，将剩下的两个 7 二进制位的数据顺序颠倒一下，因为用 Varint 方法表示时，低位的字节在前。最后，将二进制表示转换为十进制数，就得到了最终的数值。

Varint 表示方法结合了 VLC 的灵活性和现代计算机体系结构的高效率，是一种很好的混合方法。它既允许我们表示可变范围内的整数，同时还对自身的数据进行了对齐以提高解码的效率，达到了双赢。

4.3.4 为数据集找到最适合的编码方法

前面介绍的各种编码方法的最大区别是，当给定符号的概率分布期望不同时，这些编码方法的表现也不同。

因此，在为数据集选择一种 VLC 编码方法时，必须要先考虑数据集的整体大小和数据范围，并计算各个符号的出现概率。如果不这样做，得到的结果可能就是，数据集的大小不但没有压缩，有可能反而比原来的数据集还大。

为了让你了解每种编码方法最终的结果差异，我们来看下表，它展示了在理想的概率设定下，符号总数相同时使用不同的方法，每个符号需要用多少二进制位来表示。

符号的数量	Elias gamma编码 在理想的概率设定下，每个符号需要的二进制位数，理想概率为$1/(2n^2)$	Elias delta编码 理想概率为 $1/[2n(\text{lb}(2n))^2]$	Elias omega编码[a] 需要编码的最大值事先并不知道，而且数值越小出现得越频繁
1	1	1	2
2	3	4	3
3	3	4	4
8~15	7	8	6~7
64~88	13	11	10
100	13	11	11
1000	19	16	16

a 想知道更多关于 Elias omega 方法的知识，参见维基百科的相应网页。



需要记住的内容如下

VLC 编码方法是根据某个数值期望的出现频率来为该值分配码字的。因此，每种 VLC 编码方法，对于数据集中的各个符号如何分布，都有相应的期望。因此，为数据集选择适当的 VLC 编码方法，关键在于使 VLC 背后的概率模型与该数据集匹配。如果偏离了这个方向，最终得到的可能会是更大的数据流。

在信息论发展的最初 15 年左右的时间里，数据压缩技术完全局限于 VLC 方法。总的来说，为了压缩数据，工程师需要做的就是找到与其数据集匹配的 VLC 编码方法，并加以适当应用。

庆幸的是，这种情况已经一去不复返了。

下一章将讨论如何仅使用便签和笔就能生成自己的 VLC。

第5章

统计编码

5.1 利用统计使数据压缩接近熵

从输入流中取一个给定的符号，VLC 方法会赋给它一个位数可变的码字。当我们将此方法应用到一个足够长的数据集上时，最终所得到的符号平均长度就会接近整个数据集的熵。当然，前提是数据集中符号的概率分布与使用 VLC 方法背后的概率模型相匹配。

不过，还是要澄清一个事实：除了少数情形外，第 4 章中所讨论的那些 VLC 方法在数据压缩的主流领域中使用得并不多。正如第 4 章提到的那样，每种编码方法都对每个符号的概率分布做了不同的假定。

考虑一下实际使用这些方法时所需面对的问题：任意给定一个数据集，需要找到最适合的 VLC 方法对其编码，这样最终生成的数据流中的符号平均长度才会接近熵的大小。不过，如果选错了编码方法，结果可能是灾难性的，压缩后的数据流的大小甚至会比原来的还要大！

因此，我们必须要计算数据流中各个符号出现的概率，然后再与所有已知的 VLC 方法相比较，以便从中找出与数据集中符号的分布最匹配的那一个。即使这样，还是存在如下可能：需要处理的数据集中符号的概率分布与现有的 VLC 方法都不能完全匹配[1]。

那么，应该怎么办呢？答案是一类被称为**统计编码**（statistical encoders）的算法。这类算

注 1：当然，一旦转换后的输入流中符号出现概率变为已知，现代的压缩工具就可以明确选择一种 VLC 方法，这样编码器与解码器就在 VLC 方法上达成了一致，问题也就解决了。

法无须将特定的整数转换为特定的码字，而是通过数据集中符号的出现概率来确定新的、唯一的变长码字。最终的结果就是，给定任何输入数据，我们都能为其构造出一套自定义的码字集，而无须去匹配现有的 VLC 方法。

如果要更准确地描述这类算法，那么“统计编码算法通过数据集中符号出现的概率来进行编码使结果尽可能与熵接近”这样的表述可能很合适。

等一等，它是否应该被称为“熵编码”

值得指出的是，有时候你会听到人们用“熵编码”来描述这些统计类型的压缩算法（维基百科就是其中一个著名的例子）。

虽然“熵编码”这个术语现在是“统计编码”的同义词，但历史上，这个术语在学术界的使用很混乱，甚至存在着误用。

熵编码第一次正式出现可能是在 O’Neal 于 1967 年发表的一篇论文中，在这篇文章中有这样一句话：

> 因此，熵编码（也可以称为“香农 – 范诺编码”或“哈夫曼编码”）的技术有两方面的用途，即给定二进制位率时增加信噪比，或者给定信噪比时减少二进制位率。

1971 年，O’Neal 发表了另外一篇论文，这次“熵编码”出现在了论文题目中，而论文正文中却出现了“**熵编码技术（哈夫曼编码或香农 – 范诺编码）**”（entropy coding techniques）这样的复数表述。然而遗憾的是，这篇文章没有给出以前的使用示例或者具体的定义。

O’Neal 试图将所有运用统计方法来确定 VLC 的编码器都用同一个术语来表示（这是有意义的），但从来没有明确表达这一点。1972 年以后，就有很多论文开始包含“熵编码”这一术语，而问题是，这些论文的作者都没有按照 O’Neal 所暗示的定义那样使用这一术语。

例如，国际电信联盟 H.82 建议书（ITU-T，1993）将熵编码定义为“任意无损的压缩或解压数据的方法”。这是一个糟糕的定义，因为这样一来，这一术语就能用来表示 **LZ77** 这样的**字典编码算法**（dictionary encoder，本书后面的章节会讨论这方面的内容）。

总结起来就是，熵编码似乎不是一个定义清楚的或者说容易区分的概念。因此，为了表述清楚起见，本书中将避免使用这一概念。我们用统计编码这一术语来定义如下的算法，该算法以数据流中符号的频率为依据，为该数据流中的各个符号分配长度可变的码字，从而使最终的输出压缩得更小。

注意，有很多人会不加区分地混用统计编码和熵编码这两个术语。当遇到这种情况时，你可以主动纠正他们，提醒他们熵编码这一术语具有模糊性（对于认真的书虫来说，这是个很不错的开场白）。

5.2 哈夫曼编码

在对与特定分布模式匹配的符号出现概率进行大量讨论之后，我们回到现实中来，回到便签上。如果在你完成了所有烦琐的计算工作之后，想看看数据流中符号的概率分布是否与现有的压缩算法相匹配，结果发现能与很多种匹配上，那么你很幸运。

然而如果没有匹配上，又该怎么办呢？

答案是，此时你需要自己去构建压缩算法。对于给定的数据集，为了产生小的、自定义的 VLC，你需要一个输入是概率列表、输出是码字的算法。

下面介绍**哈夫曼编码**。

哈夫曼编码可能是生成自定义 VLC 最直接、最有名的方法。给定一组符号及其出现频率，该方法能生成一组符号平均长度最短的 VLC。它的工作原理是将数据排序后建立**决策树**（decision tree），然后从“树干”一直往下直到“树叶”为止，并记录下所做的是 / 否选择（几乎又回到了“20 个问题”这个游戏上）。这里所说的“树叶”就是我们要找的各个符号。

> **香农 – 范诺编码**
>
> 香农 – 范诺编码虽然在大多数的现代系统中没有太大价值，却是最早通过符号及其出现概率来生成 VLC 的方法之一。其价值之所以不大，是因为它没有达到最短的码字长度预期，但它已经很接近了。这一方法最初由香农于 1948 年提出，随后他将它告诉了罗伯特·范诺（Robert Fano），范诺后来将它作为技术报告正式发表了。
>
> 举一个应用的例子，PKZIP/IMPLODE 格式就没有使用哈夫曼编码，而是使用了两到三棵香农 – 范诺树（Shannon-Fano tree）。

5.2.1 构造哈夫曼树

简单来说，哈夫曼发现如果使用二叉树，就能利用符号表中的概率与二叉树的分支来创建优化的二进制代码。

我们来看一个不完整的小符号表，如下表所示。

符号	出现的频次
A	4
B	1
C	1

先将符号及其出现的频次写在便签纸上，再根据频次对符号进行排序，然后自下而上建一棵二叉哈夫曼树，并称它们为哈夫曼树的叶子（如图 5-1 所示）。

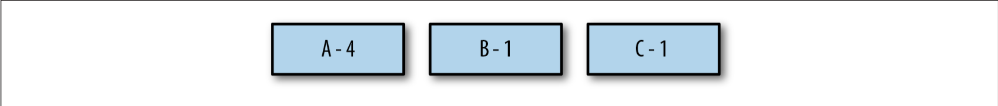


图 5-1：根据出现频率对初始结点进行排序

接下来，选择出现频次最小的两个符号，并将它们往下移一层，然后拿出新的便签将两者合并作为它们的父节点，上面写上两个符号的组合及频次相加之和，如图 5-2 所示。

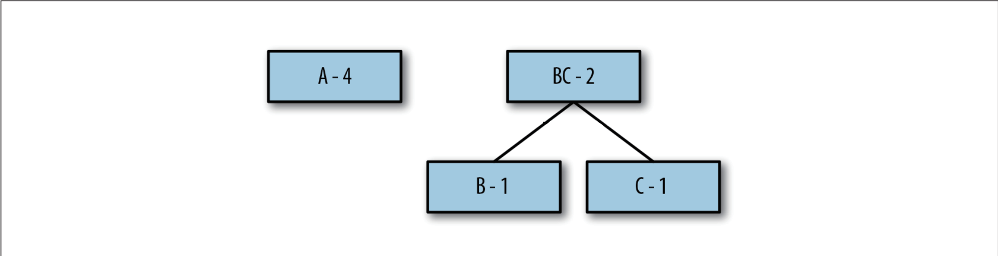


图 5-2：合并 B 和 C 创建一个父节点

然后重复上述过程，再合并剩下的 A 结点和 BC 结点创建一个新的根节点 ABC，这样它就表示了完整的符号集（见图 5-3）。

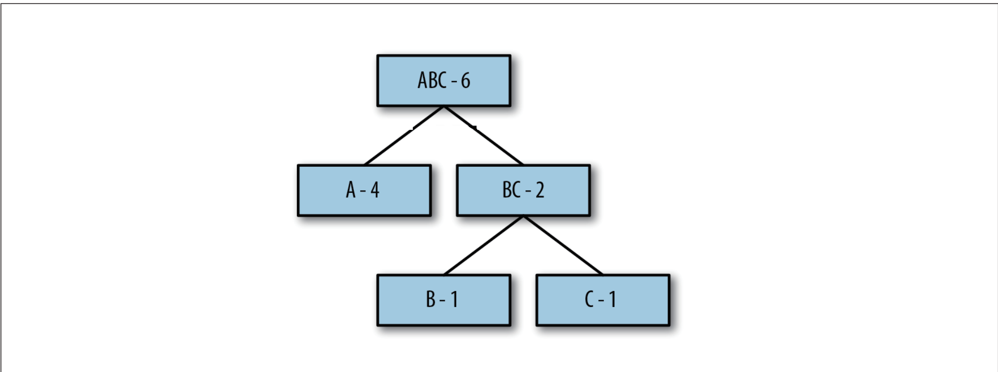


图 5-3：合并结点 A 和 BC，创建以 ABC 为根节点的树

5.2.2　生成码字

如前所述，二叉树创建好了，为了能生成码字，给所有左子树赋值 0，为所有右子树赋值 1（见图 5-4）。

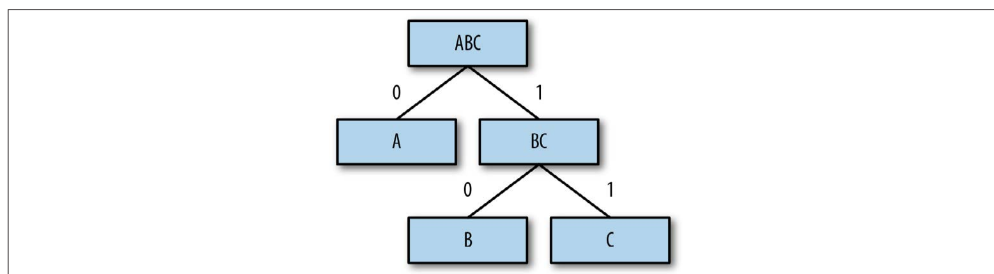


图 5-4：为左右子树分别赋值 0 和 1

最后，为了获得给定符号（叶子节点）的码字，需要从根节点“沿着树枝”往下走[2]，并将所得的 1 和 0 按从 MSB 到 LSB 排列起来，也就是从左排到右。

例如，为了确定符号 B 的码字，从根节点开始，向右遍历（得到 1），然后再向左遍历（0），这样得到的 10 就是 B 的码字（见图 5-5）。

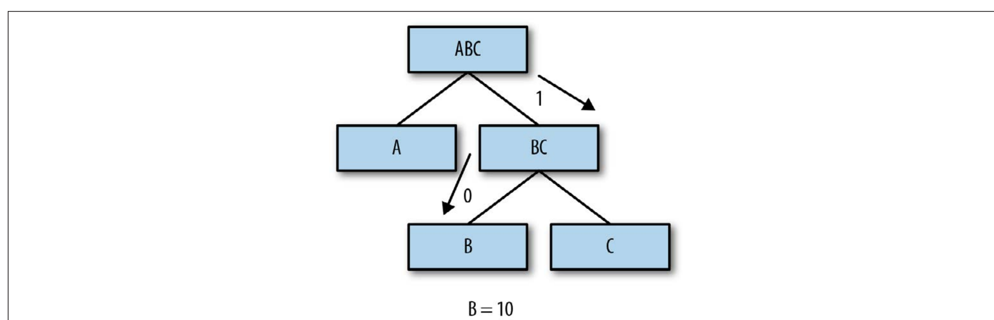


图 5-5：遍历树以获得符号 B 的码字

为了得到其他剩余符号（叶子节点）的码字，只要重复上面的过程就可以了（见图 5-6）。

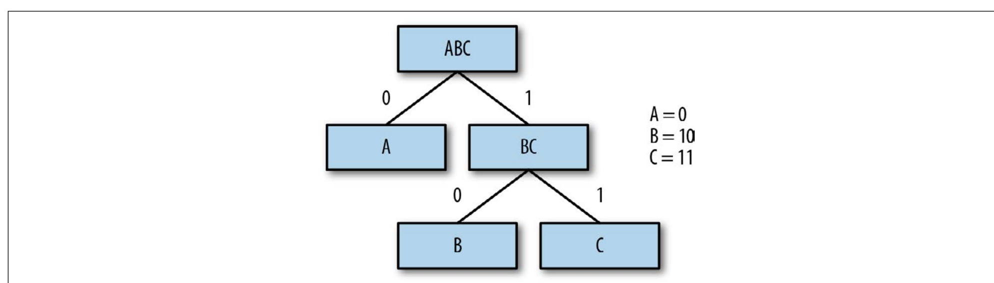


图 5-6：树中所有符号（叶子节点）的码字

注 2：这里我们抱歉地承认，这样说不完全准确。传统的哈夫曼编码其实是自下而上遍历的（与香农－范诺编码自上而下遍历刚好相反）。不过对于程序员来说，实际上自上而下遍历的代码实现会更高效，因为这样子树指针的保存更容易，树的遍历也更简单。

5.2.3 编码和解码

恭喜你！现在你已成功地为数据集构造了 VLC，可以用它们进行编码了。只要遍历输入流，对遇到的每一个符号，写下其对应的码字作为输出就可以了。

与其他 VLC 一样，为了解码，也需要将符号码字对应表与压缩后的内容放在一起传输，然后再按照标准的 VLC 的解码过程进行解码。

由于创建哈夫曼树（需使用计算资源）要比传输符号码字对应表（会增加数据流大小）困难得多，因此总是应该将码字对应表加在数据流的前面，而不是在解码时再重新创建一次[3]。

5.2.4 实际的实现方法

在过去的 50 多年里，人们已经对哈夫曼编码进行了大量的分析，不但产生了各种能在特定性能或内存阈值下工作的变体，也有针对特定概率分布的各种变体。关于哈夫曼算法及其复杂性和优化方法，已经出版了不少图书。

不过，要说的也只有这么多了，因为我们相信当你试着去使用这些算法时，你已经掌握了足够多的原理知识去理解数据与算法如何交互。

走近哈夫曼

戴维·艾伯特·哈夫曼（David Albert Huffman，1925 年 8 月 9 日—1999 年 10 月 7 日），是计算机科学领域的一位先驱，因发明哈夫曼编码而闻名于世[4]。

1951 年，哈夫曼还是一名电气工程专业的研究生，在信息论这一课程的结课考核中，他和同学们可以选择写学期论文或参加期末考试。教这门课的教授罗伯特·范诺布置了一个乍看似乎很简单的问题作为学期论文：要求学生们找出最有效的使用二进制编码表示数字、字符以及其他符号的方法。除作为智力训练之外，如果能找出这样的编码方法，也将使得对通过计算机网络传输的信息或计算机内存中存储的内容进行压缩成为可能。

哈夫曼研究了这个问题几个月，也想出了几种方法，但无法证明哪种方法最有效。最后，他对找出解决方法绝望了，决定开始学习准备期末考试。就在他准备把研究笔记扔到垃圾堆的时候，答案却在他的脑海中出现了。“那可以说是我人生中最奇特的时候，”哈夫曼说道，“突然有灵光闪现。”[5]

注 3：在本书的末尾，我们会回到如何平衡数据传输与计算资源这一话题，它的确是一个会经常出现的挑战。

注 4：他同时也是折纸数理学（mathematical origami）这一领域的先驱之一，如果你想了解这一领域，可以从 Robert Lang 的 TED 演讲 *The Math and Magic of Origami* 开始探索之旅。

注 5：出自 Inna Pivkina 的著作 *Discovery of Huffman Codes*。

那一刻的顿悟，让哈夫曼跻身于众多有突出贡献的工程师之列（相比而言，哈夫曼是幸运的，我们都知道他，而很多这样的工程师却没有留下名字），正是他们的创新性思维奠定了各种现代生活设施的技术基础。具体到哈夫曼，这些现代设施包括传真机、调制解调器以及无数其他设备。“哈夫曼编码可以说是计算机科学和数据通信领域内人员一直都在使用的基本思想之一”，斯坦福大学教授、多卷本著作《计算机程序设计艺术》的作者高德纳这样评价道。

与其他早期的重要发现一样，如果没有老师的帮助（范诺教授已经注意到克劳德·香农也在努力解决同样的问题），哈夫曼可能永远也不会发现这样的解决方法。“那应该说是我的幸运，在正确的时间去研究那个问题，并且我的老师也很好心，没有告诉我其他能力很强的人也在努力解决这个问题而使我灰心。”哈夫曼说道。

5.3 算术编码

哈夫曼编码简单、高效，也能为单个的数据符号生成最佳的码字。然而，对于给定的符号集来说，它并非总是生成最有效的码字。

事实上，哈夫曼编码能生成理想 VLC（即码字的平均长度等于符号集的熵）的唯一情形是，各个符号的出现概率等于 2 的负整数次幂（即是 1/2、1/4 或 1/8 这样的值）。这是因为哈夫曼方法会为给定符号集中的每个符号都分配一个整数二进制位长的码字。

信息论告诉我们，如果一个符号的出现概率为 0.4，那么它最理想的码字长度应该是 1.32 个二进制位，因为 $-\mathrm{lb}(0.4) \approx 1.32$。而哈夫曼方法分配给这个符号的码字长度则只能是 1 个二进制位或 2 个二进制位。

不幸的是，只要分配给一个码字的二进制位长度是整数，那么实际编码的二进制位长度与根据熵计算所需要的二进制位长度之间的差值就会比较大。要解决这个问题，我们必须放弃按照 1∶1 的比例为每个符号分配一个整数二进制位长码字的方法。

这正是**算术编码**发挥作用的地方，与按照 1∶1 的比例为每个符号分配一个码字不同，算术编码算法会将整个输入流转换为一个长度很长的数值，而它的 lb 表示则与整个输入流真正的熵值很接近。

算术压缩的神奇之处在于，它将转换应用到整个源数据上以生成一个输出值，而表示这个输出值所需要的二进制位数比源数据本身少。

故事时间：算术编码源于何处

早在 20 世纪 60 年代初，Peter Elias 就首先提出了算术压缩背后的概念（即算术编码），但是直到 10 年以后，才由 IBM 公司的 Jorma Rissane 针对其实现发表了第一个有效的研究，随之而来的还有相应的专利。

因此，在随后的几十年里，由于 IBM 公司采用了激进的专利保护战略，算术压缩几乎从算法“地图”上消失了。由于专利的问题极难解决，再加上算术压缩又非常好用，最终在 1979 年人们又发明了一种被称为**区间编码**（Range Coding）的方法。它所做的事情与算术编码基本相同，却不受算术编码相关专利的约束。

到了 21 世纪初，算术编码的专利终于到期了。算术压缩的方法再次迎来了春天，并且赢得了统计编码当前阶段标准算法的地位。

事实上，现代主流的文件、音频和视频的压缩格式（如 LZMA 和 BZIP 这样的文件格式，JPEG、WebP、WebM 和 H.264 这样的音视频格式），在统计编码步骤上都会使用算术编码压缩方法。

5.3.1 找出正确的数

算术编码的工作原理是将字符串转换为一个数，与字符串相比，表示这个数需要的二进制位数要少一些。例如，字符串“TOBEORNOT”可以用数 236 712 [6] 来表示，而 ceil(lb(236 712)) = 18，即只需要 18 个二进制位就能表示。相比而言，如果用 ASCII 码来表示“TOBEORNOT”，则需要 56 个二进制位。

不过，事情并不像前面举的例子那样简单，随机挑选一个数就可以了。相反，算术编码会根据输入流，通过一系列复杂的步骤计算出那个数。选择这个数的诀窍，实际上是对第 2 章介绍的二分查找算法进行了改进（你知道的，第 2 章是不容错过的一章）。

不妨回忆一下第 2 章，在判断某个数是比中枢值大还是小时（即需要从左右 2 个空格中选择一个），我们可以通过二分查找输出的 0/1 记录查找过程中所做的否 / 是决定。然而，如果需要从 4 个空格中选择呢？那么，我们所做的每个决定都将输出 2 个二进制位（即结果的 4 种可能取值分别表示数值范围的四分之一）。这还是很有道理吧？

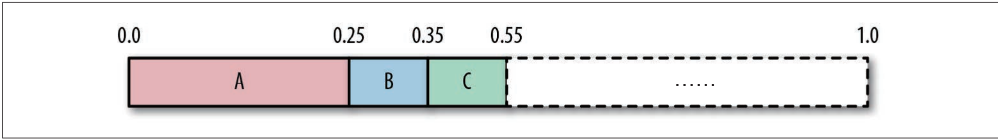
算术编码也是同样的工作思路，但同时也有一些重要的改进。

算术编码首先会创建 [0,1) [7] 这样的数值区间，然后再通过数据流中符号出现的概率对这一区间进行细分。比如说，如果 A 出现的概率为 25%，那么为 A 分配的区间就是 [0,0.25)，

注 6：事实上并不是 236 712 这个数，这里只是随便举个例子来展示一般的步骤。

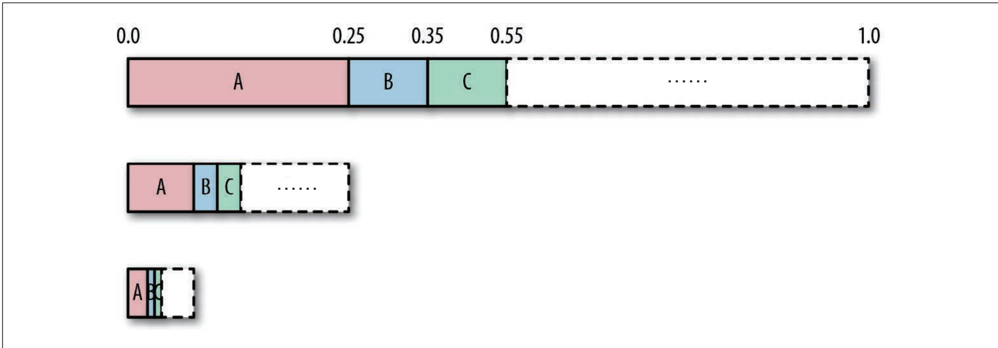
注 7：[0,1) 这样的区间记法，包含 0 在内而将 1 排除在外，使得这个区间包含从 0~0.999 99... 的所有数值（这里，9 的个数可以取任意大）。

B 出现的概率为 10%，那么 B 对应的区间则为 [0.25,0.35)，以此类推，如下图所示。



当编码器读取一个符号时，它就会找到这个符号对应的区间。例如，如果读取字符 A，那么用到的区间就是 [0.0,0.25)。在读取一个符号以后，编码器将继续细分这个取值区间，并按符号的出现比例进行细分。

例如，如果遇到的输入流中有 3 个 A，那么编码器将对 A 的取值区间细分 3 次，如下图所示。



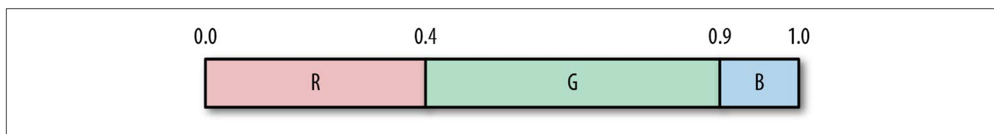
总的来说，每个符号都会以递归的方式再次细分取值区间直到读完整个输入流为止。之后，就得到最终的取值区间，比如 [0.253 212,0.254 21)。而最终输出的数值，也在这一取值区间内。例如，上面所举的输入为 AAA 的例子，其最终输出值就在 [0,0.015 625) 这个区间之内。

5.3.2 编码

下面举一个完整的编码例子，假定有 3 个字符 R、G 和 B, 其出现概率分别是 0.4、0.5 和 0.1。根据这 3 个字符的出现概率，我们在 [0,1) 范围内为它们分配了相应的取值区间，如下表所示（注意，这个表对应的熵值为 1.36）。

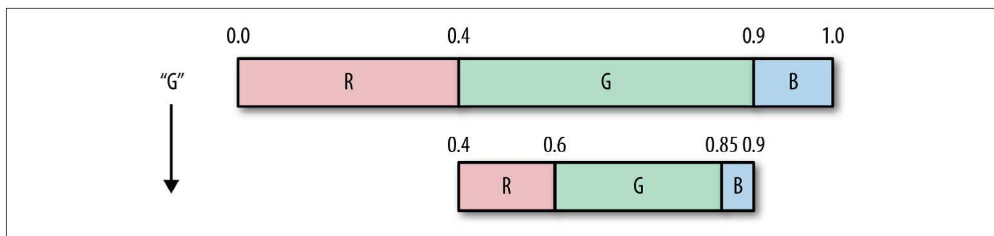
符号	概率	取值区间
R	0.4	[0,0.4)
G	0.5	[0.4,0.9)
B	0.1	[0.9,1)

下图展示了同样的信息，只不过使用了数轴线这样的方式来表示字符 R、G 和 B。



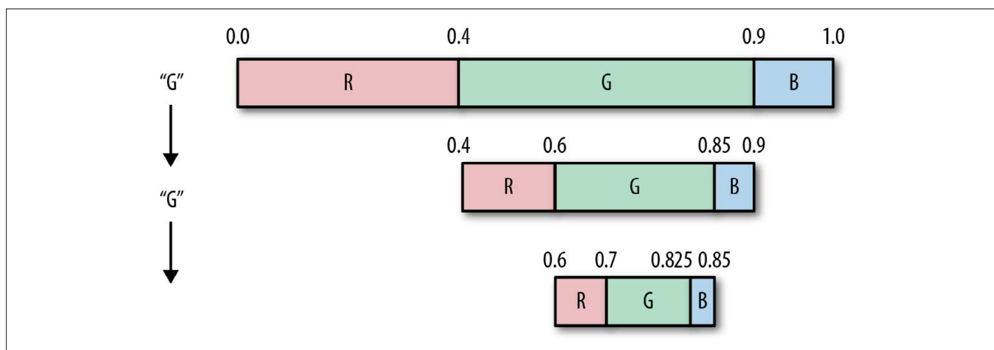
使用这样的概率设定，对字符串“GGB”编码。

从输入流中读到的第一个字符为 G，根据上表可以知道 G 的取值区间为 [0.4,0.9)。因此，需要按 3 个字符的出现概率对 [0.4,0.9) 这一区间进行再次细分，得到 3 个符号的新取值区间，如下图及下表所示。



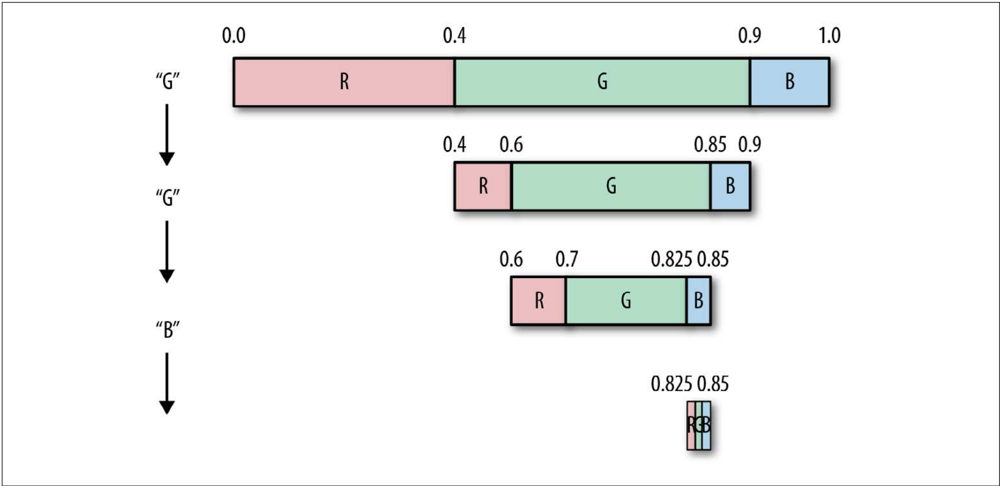
符号	概率	更新后的取值区间
R	0.4	[0.4,0.6)
G	0.5	[0.6,0.85)
B	0.1	[0.85,0.9)

接着从输入流中读取第二个字符，还是 G，因此需要继续细分 G 的取值区间，此时其范围为 0.6~0.85，并再次根据概率更新各个符号的取值区间，如下图及下表所示。



符号	概率	更新后的取值区间
R	0.4	[0.6,0.7)
G	0.5	[0.7,0.825)
B	0.1	[0.825,0.85)

读取第三个字符 B，再次细分其取值区间，并更新相应的图和表。



符号	概率	更新后的取值区间
R	0.4	[0.825,0.835)
G	0.5	[0.835,0.8475)
B	0.1	[0.8475,0.85)

恭喜你！你成功地用算术方法对一个输入流进行了编码。

5.3.3　选择正确的输出值

至此，细分完了取值区间，但要输出什么值作为最终的结果呢？

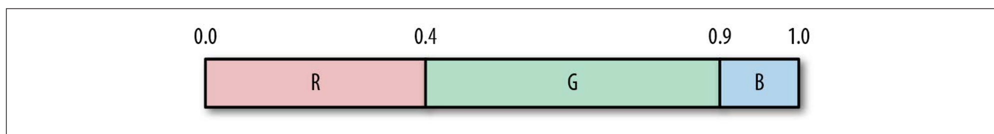
在上面所举的例子中，B 的最终取值区间是 [0.825,0.85)，在此区间内的任何数值都能让我们重新构建出原来的字符串，因此可以从中任取一值。由于本书讨论的是数据压缩，因此我们将尽可能选择可以用最少的二进制位来表示的数值（因而也尽可能与 1.36 个二进制位这个熵目标接近）。

在这个取值范围内需要最少二进制位表示的数是 0.83。

最后，由于解码器事先“知道”这个数肯定是一个小数，因此可以去掉小数点以节省空间，将最终的值确定为 83。由于 lb(83) = 7，因此平均每个字符占用 2.33 个二进制位。

5.3.4　解码

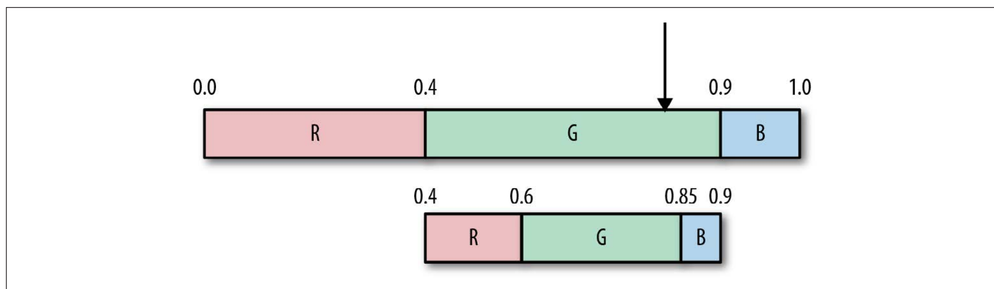
从最终输出值解码非常简单，基本是编码的逆过程。与编码相同，我们也会在 [0,1) 根据字符的出现概率等比例地创建取值区间，如下图所示。



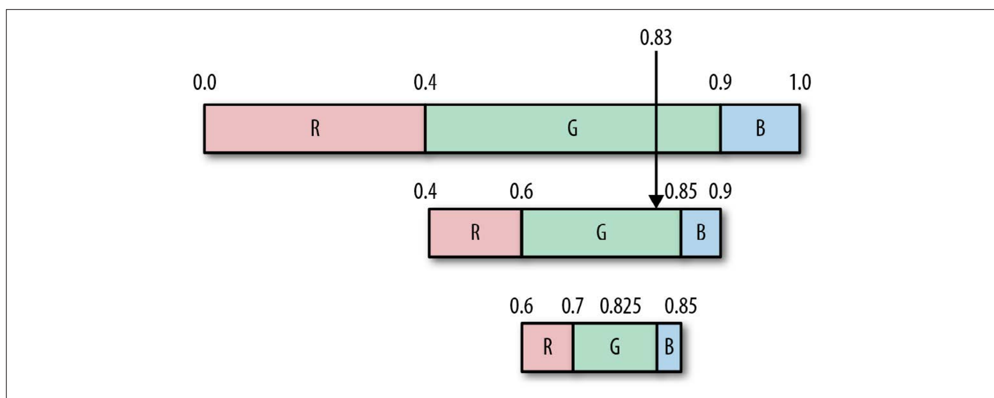
当拿到输入值 83 后，移动小数点将其变成小数 0.83，这样就能看出它落在哪个区间内，然后再将与此区间相关联的符号输出。

具体到本例中，0.83 处在 0.4~0.9，因此首先输出字符 G。

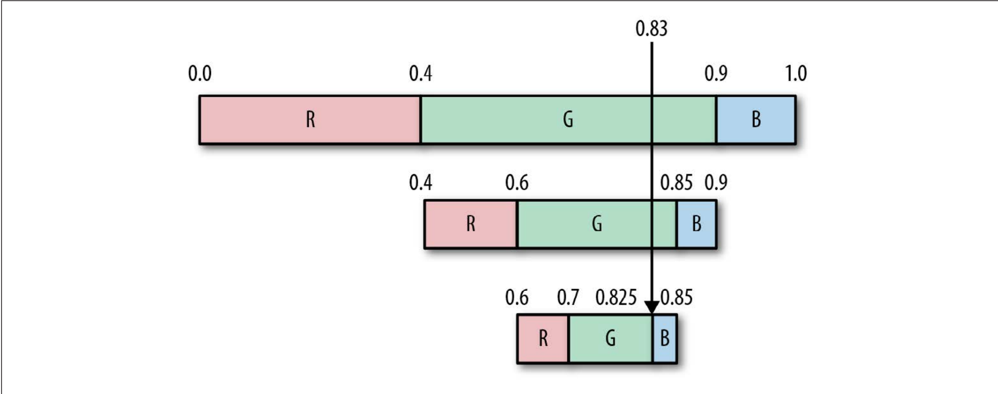
然后，再根据各符号的概率分布（与编码时相同）将 G 的取值区间细分，如下图所示。



为了得到第二个符号，需要重复这个过程。现在输入值仍然是 0.83，再次落在 G 的取值区间内，因此输出第二个 G，并再次对 G 所在的区间进行细分，如下图所示。



继续这个过程，0.83 处在 0.825~0.85，这是符号 B 的取值区间，因此接着输出 B，如下图所示。



这样就得到了最终解码后的输出流 GGB。

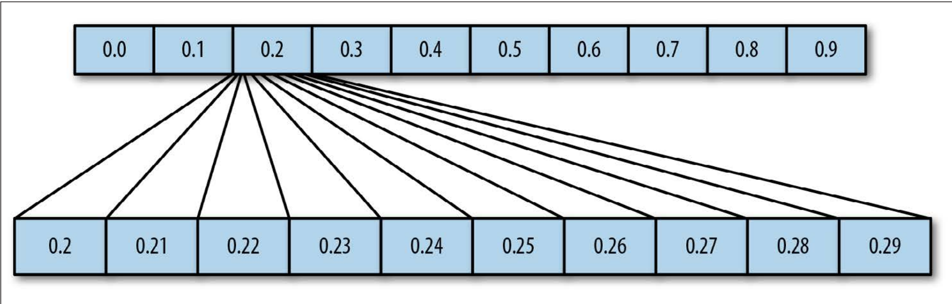
怎么样？结果很漂亮吧。整个解码的过程，基本上就是在递归的区间内画线，然后再输出在那个时候输入值所在区间对应的符号。

怎样在数据压缩中根据概率分布细分取值区间

不妨以将区间 [0,1) 均分为 10 等份作为初始状态，这其实与输入流中所有的字符都以相同的概率 0.1 出现等价，如下图所示。

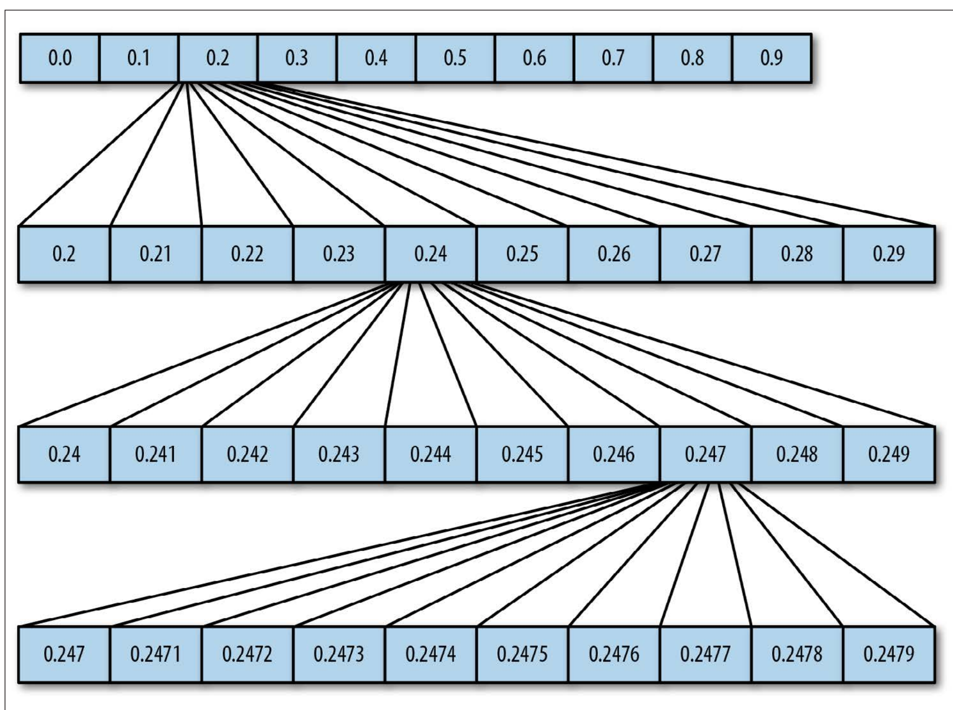


接着，读取一个字符，并将区间 [0.2,0.3) 再次分成 10 等份，如下图所示。



现在，暂停一下。我们已经读取了一个字符，如果此时停止编码的话，就会选择一个 0.2*X* 范围内的数作为输出，它会有两位小数。

随着这种模式的继续，每细分一次，最终输出的数就会增加一位小数。因此，经过三次细分会得到 0.*XXX* 这样的数，经过四次则会得到 0.*XXXX* 这样的数，如下图所示。

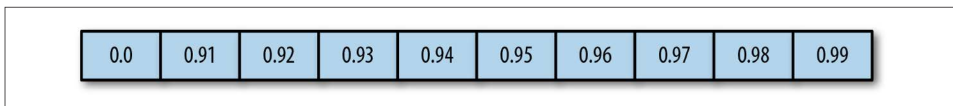


这里的问题是，由于每细分一次输出的结果就会增加一位小数，因此最后的结果是，每读取一个字符输出结果就会增加一位小数。

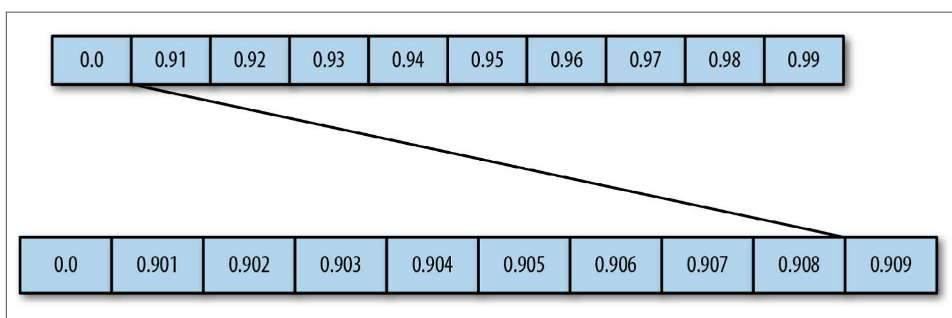
然而即使这样，还是能达到压缩的目的。例如，如果读取的是用 ASCII 码表示的数据，那么每个符号都需要 8 个二进制位，而每次细分只会让输出结果增加 lb(10)，即大约 3.3 个二进制位（因为每一步都是以 10 的次幂形式在增加的）。

不过，我们可以做得更好。

可以不等分区间，而是让某些区间的范围更大。例如，可以让第一个区间的范围为 [0,0.91)，而其他 9 个区间则在 [0.91,1) 内。其实，这与输入流中某一个符号出现的概率比其他 9 个符号大得多的情况等价，如下图所示。



读取第一个符号之后，按同样的权重继续细分 [0.0,0.91)，如下图所示。



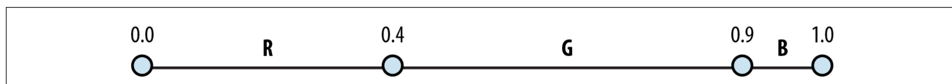
此时，如果停止编码，那么输出就可能是一位数“0”或者三位数“90*X*”。而如果将个位数赋给最可能出现的字符，就又节省了大量的空间。

提高解码性能

值得指出的是，对关注性能的人来说，上面用图例说明的解码过程可能并没有你希望的那么高性能。如果遇到 1 MB 大小的数据流，那么每次都需要将取值区间再细分为不同的符号数那么多份，如此一来则涉及很多的浮点数乘除运算。

还有一个替代的方法，就是不再细分区间，因而可以将与区间相关的影响从代码中移除。

例如前面所举的例子，其最终输出值为 83，而根据频率与区间表，可以创建如下的数值区间图。



解码时，输入值是 0.83, 位于 0.4~0.9，因此可以得到第一个输出符号 G。

不过，现在需要消除第一个符号 G 对相关数值的影响。用输入值减去 G 取值的下限（即 0.4），再除以 G 的取值区间的长度（即 0.5），其结果为 0.86，计算过程如下所示：

$$(0.83-0.4)/0.5=0.86$$

继续解码，0.86 位于 0.4~0.9，因此下一个输出符号还是 G。与前面相同，需要再次消除第二个符号 G 对相关数值的影响，用 0.86 减去 G 取值的下限（即 0.4），再除以 G 的取值区间的长度（即 0.5），其结果为 0.92。

$$(0.86-0.4)/0.5=0.92$$

而 0.92 位于 0.9~1.0，由此得到第三个符号 B。

将第三个符号输出，就完成了整个解码的过程。

5.3.5 具体实现

21 世纪初算术编码的专利到期之后，算术编码得到了大量的使用，并变得流行起来，同时也产生了大量与其具体实现相关的话题。让人印象深刻的是，人们对不少实现方法进行了修改以适应特定的**编码解码器**，如 JPG 和 H.264 编码解码器所使用的二进制版本（binary-only versions）。虽然这些**有损压缩**方法不在本书的讨论范围内，但如果你想了解更多的话，参见文章 *Practical Implementations of Arithmetic Coding*。

5.4 ANS

在哈夫曼编码与算术编码进行了将近 40 年的较量之后，这两种方法似乎都要被一种全新的统计编码算法替代。

2007 年，Jarek Duda 引入了一种新的与数据压缩有直接关联的信息论概念：**非对称数字系统**（asymmetric numeral systems，ANS）。实际上，ANS 是一种新的精确熵编码方法，所得到的结果可以和最优熵任意接近，它的压缩率与算术编码接近，而性能则与哈夫曼编码相当。

虽然 Jarek Duda 的论文详细说明了很多很酷的数学发现，但实际上我们可以像使用其他统计编码算法那样使用这一算法[8]。

(1) 根据符号的出现频率对数值区间进行细分。
(2) 创建一张表，将子区间与离散的整数值关联起来。
(3) 每个符号都是通过读取和响应表中的数值来处理的。
(4) 向输出流中写入可变的二进制位位数。

这一算法独有的两个部分在第 (2) 步（子区间与整数值关联的表）和第 (4) 步（可变的二进制位位数）中。

下面来好好地看一看。

5.4.1 通过转换表来编码和解码

tANS 是 ANS 的一种变体，它是围绕着一张表格工作的。

我们来看一个例子，假定有下表。暂且忽略这张表是如何创建的，稍后会讨论这个问题。

注 8：特别是 tANS 变体，它的设计目标就是要直接替代哈夫曼编码，参见论文 *The Use of Asymmetric Numeral Systems as an Accurate Replacement for Huffman Coding* 或者 *Asymmetric Numeral Systems: Entropy Coding Combining Speed of Huffman Coding with Compression Rate of Arithmetic Coding*。

状态/行号	A	B	C
1	2	3	5
2	4	6	10
3	7	8	15
4	9	11	20
5	12	14	25
6	13	17	30
7	16	21	
8	18	22	
9	19	26	
10	23	28	
11	24		
12	27		
13	29		
14	31		

给定了这张表，我们来看一下如何对输入的字符串 BAA 进行编码。

(1) 给定输入字符串 BAA，需要从 [行号 , 输入符号] 对开始编码，这里就是 [1,B]。[9]
(2) 有了这样的 [行号 , 输入符号] 对，就能定位到下一个需要读取的值在表格中的位置。通过查表，得到单元格 [1,B] 的值是 3，因此，下一个 [行号，输入符号] 对的值为 [3,?]。
(3) 为了得到列号，接着读下一个符号，为 A，得到新的表位置 [3,A]。
(4) 再次查表，得到单元格 [3,A] 的值是 7。
(5) 继续读下一个符号，也为 A，得到新的表位置 [7,A]。
(6) 表中此位置对应的值为 16。
(7) 可以一直这样操作下去，直到表尾（或者读取字符串的最后一个字符）。
(8) 根据这张表，将输入的 BAA 转换为了 [3,7,16]。

解码过程则与编码过程相反。

(1) 从最后一个值 16 开始解码。
(2) 搜索整个表后，发现 16 位于表的 7 行，A 列。
(3) 将 A 输出作为符号，而 7 则变为当前的值。
(4) 再次搜索表，发现 7 位于表的 3 行，A 列。
(5) 再次将 A 输出，此时 3 则变为当前的值。
(6) 在表中搜索 3，发现它位于 1 行，B 列。由于已经到了第 1 行，所以不能继续操作，解码过程结束。

注 9：注意，1 始终是初始状态。

(7) 解码后的符号流为 [A,A,B]，这是因为**从后往前**解码。
(8) 倒置所得到的字符串，就得到了源字符串 BAA。

从上面的编码和解码过程可以看出，有了这张表以及相应的操作方法，就可以对输入的字符串正确地编码和解码。

5.4.2 创建备查表

正如前面我们所看到的那样，这一算法的核心就是这张神奇的备查表，它使得从符号转换为数值再从数值转换为符号成为可能。创建这张表时，需要先根据符号出现概率的大小排序，每个符号作为表的一列，从左往右概率依次递减。

前面的那张表中，符号 A、B、C 的概率 $P([A,B,C]) = [0.45,0.35,0.2]$, 其中每个符号都被分配作为表的一列。

接下来需要往表中填入数值，这些数值需满足以下条件：

- 表中的每个值都是唯一的（即不存在重复）
- 每列都按照值从小到大排序
- 每行的值都要比该行的行号大

如果能遵循这些原则，那么前面展示的编码 / 解码转换就都能正常工作，但是如果想要 tANS 变成真正的熵编码器，还必须要考虑以下两个性质：

(1) **在确定每一列值的个数时**，需满足该值乘以 maxVal[10] 后，等于该列符号的出现概率；
(2) **在确定每一行的值时**，需确保该行列选择的值与该列符号的出现概率一致，这样当用该值除以行号，所得商就会（近似）等于该列符号的出现概率。

下面来好好看看前面那张表的这些性质。

第一个性质比较简单。这张表中最大的值 maxVal = 31。为满足前面所述的性质，我们需要将 maxVal 细分，并将 $P(S) \times$ maxVal 个值分配给各个列。具体到前面所举的例子，有如下计算。

- 符号 B 的出现概率 $P(B) = 0.35$，因此 B 列会有 $\text{floor}(0.35 \times 31) = 10$ 个值出现。
- 符号 C 进行同样的处理，因此 C 列会有 6 个值（$0.2 \times 31 = 6$）。
- 最可能出现的符号，也就是最左边的 A 列，一共有 $P(A) \times \text{maxVal} + 1 = 0.45 \times 31 + 1 = 14$ 行，这是因为 A 列还需要为 maxVal 增加额外的一行。

注 10：我们还没有说 maxVal 是什么意思，不过很快就会选择一个合适的值作为 maxVal。

这样，就对 maxVal 进行了细分，让每个符号出现的行数与 maxVal 的比等于其出现概率。[11]

第二个性质表明，每一行的值可以用行号乘以每个符号的出现概率计算出来。

具体到前面所举的样表，符号表 S = [A,B,C] 中各符号的出现概率为 P(S) = [0.45,0.35,0.2]，再任意选一个行，比如说第 5 行，从左到右各单元格的值依次为 12、14、25。

现在，用这些值除行号，所得的结果与符号的出现概率非常接近：[5/12,5/14,5/25] = [0.416 67, 0.357,0.2] ≈ P(S)。

第二个性质必须对每一行都成立，通过下表可以看出对 A 列来说，每一行的值基本都能使 P(A) = 0.45。

行号	P(A)	行号/P(A)	表中实际值	行号/值
1	0.45	2.2223...	2	0.5
2	0.45	4.4444...	4	0.5
3	0.45	6.6666...	7	0.42
4	0.45	8.8888...	9	0.444...
5	0.45	11.1111...	12	0.416...

那么，为什么有时候表格中实际分配的值会和计算得出来的值不一致呢？答案是为了避免重复。

表中的每个值都必须是唯一的，但由于计算的时候需要四舍五入，因此有时候就难免会出现彼此冲突的情况。例如，1/P(C) = 5，而 2/P(B) = 5.714，与 5 接近。

可以通过使用表中还没有使用的下一个更大的值，来解决冲突问题。例如，对前面的样表 B 列、2 行这个单元格赋值时，就不能再用 5（因为它已经在 1 行使用了），此时可以尝试使用下一个更大的值 6，而 6 还没有使用过，因此可以使用 [12]。

如果这两条性质都能满足，那么我们就能完整地创建出前面那样的表，对数据正确地进行编码解码从而实现压缩的目的。

选择一个 maxVal

maxVal 的选择直接影响到输出的压缩结果，而压缩结果又直接与编码所允许的整数精度相关。

因此，目标就是为每个符号分配一个子区间，使其长度与该符号的出现概率相匹配。如果编码过程和表中值的计算都是在浮点数范围内进行，那么这不会是什么大问题：让每个符

注 11：注意，如果创建的是一张完整的二维表（也就是说每一列的高度都相等），就需要在每一列超出行高的那些位置上填一些特定的值（比如 –1 或者其他值），以表明这些单元格是空的或无效的。

注 12：如果考虑性能，其实有很多种不同的方法可以对已使用的值进行标记。建议使用 Andrew Polar 的“宾果板”方法（“bingo board” method）。

号对应区间的大小等于其概率就可以了。然而现在的问题是，编码与解码过程都是在和整数打交道。因此，需要创造出某个大小的整数空间（即从 2 到 maxVal），这样可以为每个符号分配相应的子空间而不会遇到精确度的问题。

假定需要处理的数据集中有 28 个不同的符号，那么最低限度下需要 LOG2(28) = 5 个二进制位的空间大小，这样一来，maxVal 的值就等于 $2^5–1 = 31$。

然而，由于每个符号的出现概率并非完全相同，这个取值并不能让我们有足够的空间为每个符号都分配不同的空间大小。为了适应这种情况，就需要增加二进制位的数量。

因此，选择的 maxVal 应该是一个函数，该函数是所需要的最小二进制位数加上由于精度的需要而额外增加的二进制位数：

$$\text{numPrecisionBits} = \text{LOG2(numSymbols)}+\text{magicExtraBits}$$
$$\text{maxVal} = (2^{\text{numPrecisionBits}}) - 1$$

这里，magicExtraBits 一般取 2~8 的某个值，或者取任何对于具体数据集来说合适的值。正如稍后会展示的那样，对于 magicExtraBits 的取值，要综合考虑压缩质量和时间，因为这个值越大，压缩率就越高，但同时压缩需要的时间也会越长。

5.4.3 使用ANS压缩数据

前面介绍了备查表的工作原理，然而，那里的输出达不到统计压缩的目的。为了达到这一目的，还需要对前面介绍的算法进行一些调整。

- 首先，不再从 1 行开始，而是将初始状态（行号）选择为 maxVal。
- 其次，对从数据流中读取的每个符号：
  - 将目标行设为该符号的列高度；
  - **右**移状态值直到它比目标行的值小；
  - 状态值右移的过程中所丢弃的每个二进制位都应该输出到编码后的二进制位流中。

还是前面那张表，经过这样的调整之后，下面来看字符串“ABAC”的编码过程。

(1) 从初始状态开始，由于初始状态值 = maxVal = 31（二进制为 11111），因此从 31 开始。
(2) 从字符串中读取第一个符号 A，因此第一个表位置为 [31,A]，同时将目标行设为符号 A 的列高 14。
(3) 由于 31 > 14，因此需要移位并将相应的二进制位输出。
   a. 将状态值 11111 右移一位，得到 1111，同时将截断的最右边的 1 输出。
   b. 现在，状态值为 15，还是比 14 大，再次右移一位，状态值变为 111，再次将右边的 **1** 输出。
   c. 状态值现在为 7，我们已经截断并输出 2 个二进制位（即 11）。

(4) 现在，将备查表单元格 [7,A] 的值 16（二进制为 10000）赋给状态值。
(5) 读取下一个符号 B，得到第二个表位置为 [16,B]，同时将目标行设为符号 B 的列高 10。
(6) 由于 16 > 10，需要移位并输出相应的二进制位。
    a. 状态值 16 右移一位后变成 8（二进制为 1000），同时将最右边的 0 输出。
    b. 由于 8 < 10，因此可以继续读取一个符号。
(7) 将备查表单元格 [8,B] 的值 22（二进制为 10110）赋给状态值。
(8) 读取下一个符号 A，得到第三个表位置为 [22,A]，同时将目标行设为符号 A 的列高 14。
(9) 由于 22 > 14，需要移位并输出相应的二进制位。右移后状态值变为 1011，同时将最右边的 0 输出。
(10) 将备查表单元格 [11,A] 的值 24（二进制为 11000）赋给状态值。
(11) 读取下一个符号 C，得到第四个表位置为 [24,C]，同时将目标行设为符号 C 的列高 6。
(12) 由于 24 > 6，需要移位并输出相应的二进制位。右移两次后状态值变为 110，同时将最右边的 00 输出。
(13) 将备查表单元格 [6,C] 的值 30（二进制为 11110）赋给状态值。
(14) 由于字符串已经为空，将状态值（11110）全部输出。
(15) 因此，最终的输出流为 11000011110，共 11 个二进制位。当然，除此之外，还需要加上符号概率表所占的二进制位数。

5.4.4 解码示例

解码的过程与上面的编码过程相反。

(1) 从压缩后的数据流中读出符号出现频率数据。
(2) 根据符号出现频率信息创建备查表。
(3) 继续从数据流中读出状态值。
(4) 找出该状态值在备查表中的位置。
(5) 将该值所在的列号作为符号输出。
(6) 将当前行号赋给该值。
(7) 继续从数据流中读取一些二进制位（并放到该值的后面，使其成为完整的状态值）。

注意，在这个例子中 maxVal = 31，即精度为 5 个二进制位（5 bits of precision）。

创建好备查表之后，我们来看一下从编码后的数据流 11000011110 解码的过程。

(1) 由于目标状态值为 5 位，因此先从数据流中读出最后 5 个二进制位，即 11110（即十进制数 30）。
(2) 找到唯一值 30 在表中的出现位置，为 [C,6]。
(3) 输出符号 C。

(4) 6（二进制为 110）只有 3 个二进制位，因此需要从数据流中再读取 2 个二进制位（因为状态值为 5 个二进制位）。

(5) 继续读取数据流的最后 2 个二进制位 00，并把它加到 110 后，现在的状态值为 11000（即十进制数 24）。

(6) 找到 24 在表中的出现位置，为 [A,11]。

(7) 输出符号 A。

(8) 11（二进制为 1011）只有 4 位，因此需要从数据流中再读取 1 个二进制位，此时状态值为 10110（即十进制数 22）。

(9) 找到 22 在表中的出现位置，为 [B,8]。

(10) 输出符号 B。

(11) 8（二进制为 1000）只有 4 个二进制位，继续读取 1 个二进制位，此时状态值为 10000（即十进制数 16）。

(12) 找到 16 在表中的出现位置，为 [A,7]。

(13) 输出符号 A。

(14) 7（二进制为 111）只有 3 个二进制位，继续读取 2 个二进制位，此时状态值为 11111（即十进制数 31）。

(15) 由于状态值现在等于最大值 maxVal（11111），我们知道这是结束的标志，因此停止解码。

(16) 将得到的字符串倒置，就得到了源字符串 ABAC。

5.4.5　压缩是从哪里来的

答案是压缩来自于逐位输出（bit-wise output）。

因为出现可能性越小的符号其列高越低，有效的行号值离最可能出现的符号也就越远（二进制位距离意义上的远），所以为了得到更小的行号，就需要进行更多次的右移操作，这也意味着每次循环会有更多的二进制位输出到数据流。因此，出现可能性越小的符号，就会输出更多的二进制位到最终的数据流中。

就像前面提到的那样，位数越多，空间的精确度就越高（因此，maxVal 的值也越大）。这也会让备查表中出现更少的精度冲突，因为更大的空间可以让整数与通过 $P(\mathrm{S}) \times \mathrm{maxVal}$ 计算出来的值更接近。回忆一下，当备查表中出现冲突时，就需要通过线性查找法去找一个更大的没有使用的值来解决冲突。在编码时，计算出来的值与表中实际值之差，就会造成为了让状态值小于目标行值而进行更多次的右移。当精确度较高时，表中的值冲突就会减少，需要人为增大的值也会减少，因此右移时输出的二进制位数也就变少了。

当然，这样做也会有不利的一面。精确度高了，就需要有更大的备查表，创建这张表需要的时间就会越长，存储表需要的空间也会越大。因此，要根据特定的情况，综合考虑性能和存储的要求权衡取舍。

5.5 在实际压缩中，选择哪一种统计压缩算法

现在，在知道了 3 种很棒的可以应用的统计压缩算法后，如果有数据集需要压缩，你会选择哪一种呢？

这是一个普遍的问题，在过去 20 多年的大部分时间里，数据压缩领域内有大量关于哈夫曼编码与算术编码的争论。1993 年，这一争论首次曝光，这一年 Bookstein 和 Klein 发表了一篇题为 *Is Huffman Coding Dead?* 的论文。

虽然这篇文章已发表 20 多年了，但争论双方依然保持原来的观点。

因为计算机变得越来越快（并且算术压缩的专利已经到期），所以算术压缩已成为目前的主流算法。它不仅应用在大多数的多媒体编码器中，甚至有了有效的硬件实现。

但 ANS 改变了一切。虽然它在数据压缩领域里出现的时间还不长，但是已开始取代过去 20 多年里占据主流地位的哈夫曼编码和算术编码。

例如，ZHuff、LZTurbo、LZA、Oodle 和 LZNA 这些压缩工具已开始使用 ANS。鉴于其速度和性能，ANS 成为主要的编码方法似乎只是时间问题。实际上，在 2013 年，这一算法又出现了一个被称为有限状态熵（Finite State Entropy，FSE）的更注重性能的版本，它只使用加法、掩码和移位运算，使 ANS 对开发人员更具吸引力。它的性能是如此强大，以至于 2015 年推出了一款名为 LZFSE 的 GZIP 变种，作为苹果下一代 iOS 版本的核心 API。

目前看来未来的路似乎很清楚：ANS 和 FSE 将终结哈夫曼编码和算术编码在压缩领域内几十年的“霸主”地位。

第 6 章

自适应统计编码

6.1 位置对熵的重要性

第 5 章介绍的所有统计编码算法，在编码开始之前都需要遍历一次数据，以计算出各符号出现的概率。这样做是有缺点的，为了计算概率总需要多遍历一次数据集，而在计算出整个数据集中各符号的出现概率后，还要继续处理这些数值。如果是相对较小的数据集，那么这些就不是什么问题。

然而，随着要压缩的数据集变大，统计编码的结果与熵的偏差也会越来越大，这是因为数据集的不同部分有着不同的概率特征。如果处理的是流数据，比如视频流或音频流，由于整个数据集没有“结尾”，因此就不能“遍历两次”。

这些概念适用于所有的流数据，下面就在相对简单的示例数据集中来看一看这些概念。在数据流中，字符 Q 可能会在前三分之一部分出现很多次，而在后三分之二部分则一次也没有出现。统计编码算法的概率表无法处理字符 Q 分布的这种局部性。如果字符 Q 出现的概率为 0.01，那么通常我们会期望它在整个数据流中均匀分布，也就是说，大约每 100 个字符中就有 1 个是 Q。

然而实际数据的情况并非如此，数据中总会存在某种类型的**局部偏态**（locality-dependent skewing）[1]，将某些符号、想法或者单词集中在数据集的某个子区间里。

结果是统计编码算法生成的编码比根据熵生成的更臃肿，这是因为其所依据的概率信息没

注 1：这个词完全是我们自造的。

有考虑统计上的局部变化。例如，如果将数据流分为 *N* 块并且每块都单独压缩，那么得到的结果可能会比将数据流整体压缩得到的结果小（如果数据流中存在很多局部偏态的情况的话[2]）。

下面来看看这样一个简单的示例数据集：

```
AAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBCDEFGHIJKLMNOPQRSTUVWXYZ
```

该数据集的熵约为 3.48，说明平均每个符号需要使用 3.48 个二进制位的空间，整个数据集压缩后的大小为 198.36 个二进制位。如果使用哈夫曼编码，编码后的大小为 202 个二进制位，也就是每个符号约使用 3.54 个二进制位[3]，这样的结果看起来还不错。

但老实说，这里其实可以做得更好。我们可以清楚地看到整个字符串的前一半高度重复，只由两个字符组成。实际上，遇到这种情况我们会去想办法分割整个字符串，这样，对字符串的前一半，就会有更好的编码方法。与将整个字符串转换为 VLC 相比，将字符串分成两半，前一半每个符号平均只需要 1 个二进制位，后一半平均需要 5 个二进制位，难道这样的结果不是更令人满意吗？分割为两半后，整个字符串共需要 122 个二进制位，平均每个符号需要 2.1 个二进制位。（这里需要指出的是，这样的结果已经超越了香农，而且远远超越。）

这让我们接触到数据压缩领域内一个重要的理论，即局部性很重要（locality matters）[4]。由于数据流一般以线性的方式生成，因此数据流中很有可能会出现某一部分的特征与其他部分完全不同的情况。

要实现这种方式的优化，真正的挑战在于如何以最佳的方式分割数据流。像前面那样先扫描，再试着合理地分段，只会让你更抓狂，并且会有一种解决 **NP 完全问题**（NP-complete problem）的感觉。因此，我们编码时不再提前扫描并去找合适的分割点，而是允许统计编码算法自动“重置”。

这一过程在概念上很简单，就是在编码时，如果“期望”的熵与“实际的符号平均二进制位数”之间出现显著差异，那么统计编码算法会重置概率表，并使用重置后的概率表进行编码。

这种具有适应数据流熵的局部特性能力的统计编码算法，通常被称为“动态”或“自适应”统计编码算法。这些算法变体构成了大多数重要的、高性能的、高压缩率的多媒体数据流（如图片、视频及音频）压缩算法的基础。

注 2：自造术语很有趣，多次使用自造的术语更有趣，因此我们会继续这样做。

注 3：从表面看,这两个值（熵与实际的符号平均二进制位数）之间的差异与第 5 章介绍的要取整数有关（特别是哈夫曼编码与算术编码之间的差异有关）。

注 4：实际是性能很重要（#PERFMATTERS），但那会是一本完全不同的书。

6.2 自适应VLC编码

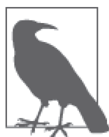
下面来看看自适应算法中最简单的一种，了解其基本的工作原理。

一般来说，统计压缩有 3 个步骤[5]：

(1) 遍历数据流并计算各个符号的出现概率；
(2) 根据概率为符号生成 VLC；
(3) 再次遍历数据流并输出对应的码字。

从上面可以看出，压缩时需要遍历（或者说扫描）数据流两次，并且整个数据集只有一套 VLC 表。这里的问题是，VLC 表是静态的。

而在**自适应**的压缩算法中，这 3 个步骤简化为仅遍历一次数据集，但是过程要更复杂。关键是符号码字对应表并非必须一成不变，相反，可以根据读到的符号更新它。



自适应统计编码的关键在于其符号码字对应表并非一成不变，相反，可以根据读到的符号动态地生成 VLC。这一过程的动态性质，让我们可以根据需要对 VLC 表进行修改，比如对其重置。

6.2.1 动态创建VLC表

动态创建 VLC 表的原理如下。

在编码器处理数据流时，每读取一个符号，编码器都会问：

- 这个符号之前出现过吗？
  - 如果出现过，那么输出当前分配的码字，并更新其出现的概率。
  - 如果没有，则进行一些特殊处理（稍后会讲到这个部分）。

请记住上面的内容。假定你正在处理某个数据流，已经知道了其中的符号及相应的概率期望。目前已有的 VLC 表如下表所示。

符号	概率	码字
A	0.5	0
B	0.4	10
C	0.1	11

注 5：当和那些非常聪明的人谈论数据压缩时，他们通常会认为统计编码只有两个步骤：建模和预测。John Brooks，看到这里，你很高兴吧？

接下来，需要从输入流中读取下一个符号，这个符号恰好是 B，于是进行下面的操作。

(1) 输出 B 当前对应的码字 10。

(2) 更新相应的概率，因为 B 出现的可能性现在变大了一些（其他符号出现的概率变小了一些），更新后的表如下所示。

符号	更新后的概率	码字
A	**0.45**	0
B	**0.45**	10
C	0.1	11

(3) 继续读取下一个符号，还是 B，因此再次输出 10，并继续更新相关符号的概率。再次更新后的表如下所示。

符号	更新后的概率	更新后的码字
A	**0.4**	**10**
B	**0.5**	**0**
C	0.1	11

注意，这里有一件重要的事情发生。由于 B 已经成为数据流中最可能出现的符号，因此将最短的码字分配给它，如果下一次读到的符号还是 B，那么相应的输出会是 0，而不是之前的 10。

通过动态更新读到的符号的出现概率，我们就可以根据需要去调整分配给各个符号的码字长度。

解码

为了确保这样的处理真正有效，下面来看解码过程。

从现成的频率以及下表开始。[6]

符号	概率	码字
A	0.45	0
B	0.45	10
C	0.1	11

从输入流中去读当前存在的码字，这里是 10，然后输出 B。由于 B 再次出现，因此需要更新概率表，更新后的表如下所示。

注 6：注意，这里我们给出起始频率只是为了教学的方便，现实中，你是不可能得到这样的信息，而需要从头来创建这张表。

符号	更新后的概率	更新后的码字
A	**0.4**	**10**
B	**0.5**	**0**
C	0.1	11

看，这张表的变化与编码时完全相同。

一切都正常工作。

只要解码器更新符号表的方式与编码器相同，那么这两张表就会始终保持同步。

下面再来确认一下。

❑ **编码**

(1) 从输入流中读取符号。
(2) 输出该符号对应的码字到输出流中。
(3) 更新符号的出现概率并重新生成码字。

❑ **解码**

(1) 从输入流中读取码字。
(2) 输出该码字对应的符号到输出流中。
(3) 更新符号的出现概率并重新生成码字。

这就是自适应统计编码算法工作的大致流程。编码器和解码器都会动态更新符号的出现概率及相应的码字，这通常以积极的方式影响压缩。

6.2.2 字面值

然而，现在我们仍然面临以下两个问题。

- 在开始编码前，最初的 VLC 表是什么样子？
- 在解码过程中，如果读到一个 VLC 表中不存在的符号该怎么办？

这两个问题实际上是同一个问题的两种问法，答案是**字面值词条**（literal tokens）。

字面值词条其实就是唯一的“假”符号，编码器和解码器将它作为从**字面值流**（literal stream）中读取符号或者将符号写入到字面值流中去的信号。字面值流，指的是只包含字面值的数据流，换句话说，就是那些实际上被编码的符号流，而且是根据其在数据流中出现的先后顺序排列的。

举个例子，如果数据流是“AAAAABCABC”，那么对应的字面值流为“LITERAL/A/B/C”，编码后的二进制流则为 00 1010 01 00 00 00 01 1011 01 1100 00 10 11，如图 6-1 所示。

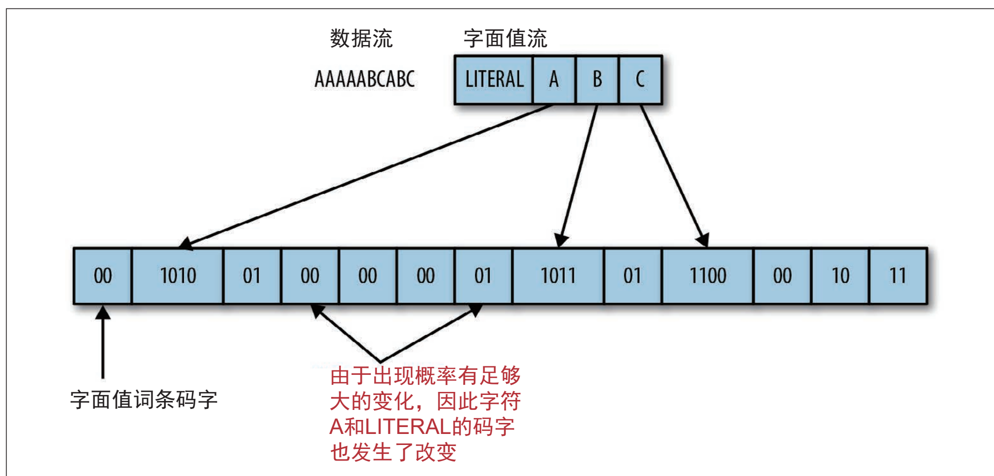


图 6-1：该图展示了从字面值流读取的字面值词条及概率变化，是如何出现在编码后的数据流中的

在编码过程中，当编码器读到一个之前没有遇到的符号时，它需要做以下两件事。

(1) 将 LITERAL 对应的码字写入输出流。
(2) 将读到的新符号添加到字面值流中。

而在解码过程中，当解码器读到一个 LITERAL 码字时，则需要做下面两件事。

(1) 从字面值流中读取下一个字面值。
(2) 将该字面值写入输出流并更新 VLC 表。

下面来看一个具体的例子。

开始编码时，我们还没有读任何符号，因此对于读到的第一个符号，需要输出一个字面值。因为这是唯一的选择，所以 VLC 表最开始只包含 LITERAL 这个符号，其出现概率为 100%，对应的码字为 00，如下表所示。

符号	概率	码字
< LITERAL >	1.0	00

当我们从输入流中读到一个新符号时，将 LITERAL 对应的码字输出，后面跟着新符号的二进制位。与前面的操作类似，接着就需要更新符号表以及各符号的出现概率。

符号	概率	码字
< LITERAL >	0.5	00
A	0.5	01

假定之后又读到符号 A，接着又读到另一个新符号 B，因此，读到的符号依次为 < LITERAL > A A < LITERAL > B，现在的概率如下表所示。

符号	概率	码字
＜LITERAL＞	0.4	01
A	0.4	00
B	0.2	10

因此，对于输入字符串 AAAAABCABC，下面是完整的编码过程示例。（可以参考图 6-1 获得一些提示。）

这些字面常量未编码时的 4 个二进制位表示分别为：

A = 1010

B = 1011

C = 1100

注意，对于 VLC，我们将仅使用 00、01、10、11 这样两位长的码字来简化说明。

(1) VLC 表中仅包含 LITERAL 这个字符，其出现概率为 1.0，对应的码字为 00。

(2) 读取第一个符号 A。

  a. 符号 A 不在 VLC 表中，因此需要输出字面值词条 LIT 的码字（即 00），然后是 A 对应的 4 个二进制位表示：1010。

  b. 将符号 A 添加到 VLC 表中，并根据符号的出现频次更新表。由于 A 和 LITERAL 各出现一次，因此两者的出现概率均为 0.5，并为两个符号分配码字：LIT = 00，A = 01。

(3) 继续读下一个符号，还是 A。

  a. 由于 A 已经在表中，因此输出其对应的码字 01。

  b. 更新 VLC 表。符号 A 已成为最经常出现的符号，因此需要重新分配相应的码字：A = 00，LIT = 01。

(4) 继续读取下一个符号，还是 A。

  输出 A 对应的码字 00 并更新 VLC 表中的概率。

(5) 继续读取下一个符号，又是 A。

  输出 A 对应的码字 00 并更新 VLC 表中的概率。

(6) 继续读取下一个符号，仍然是 A。

  输出 A 对应的码字 00 并更新 VLC 表中的概率。

(7) 继续读取下一个符号，是 B。

  a. B 不在 VLC 表中，因此需要输出字面值词条 LIT 的码字（即 01），然后是 B 的 4 个二进制位表示：1011。

  b. 将 B 添加到 VLC 表中并更新表，A = 00，LIT = 01，B = 10。

(8) 继续读取下一个符号，是 C。

  a. C 不在 VLC 表中，因此需要输出字面值词条 LIT 的码字（即 01），然后是 C 的 4 个二进制位表示：1100。

b. 将 C 添加到 VLC 表中并更新表，A = 00，LIT = 01，B = 10，C = 11。

(9) 继续读取下一个符号，是 A。

输出 A 对应的码字 00 并更新 VLC 表中的概率。

(10) 继续读取下一个符号，是 B。

输出 B 对应的码字 10 并更新 VLC 表中的概率。

(11) 继续读取下一个符号，是 C。

输出 C 对应的码字 11 并更新 VLC 表中的概率。

因此，最终的输出流为 00 1010 01 00 00 00 01 1011 01 1100 00 10 11（见图 6-2）。

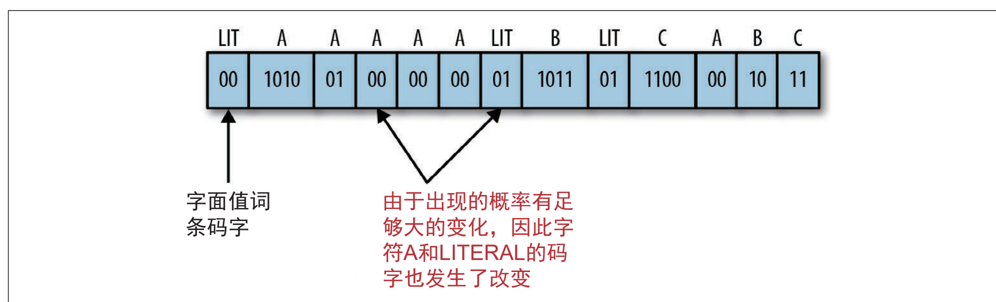


图 6-2：最终编码后的输出流

那么接下来，你能用相反的步骤解码吗？

6.2.3　重置

自适应统计编码的真正强大之处在于，当输出流的熵要变大失控时，它能重置输出流。

以 [AAABBBBBCCCCCC] 为例，并将符号放在尖括号中来表示相应的字面值，如 < A > 。

对上述字符串编码后，得到如下的输出：

< A > ,0,0, < B > ,1,1,1,0, < C > ,11,11,11,11,0

注意，最后的码字 0 对应的是最后一个符号 C，此时 C 的出现概率已经足够大，因而需要重新为其分配码字。从中可以看到，更多的符号以及特定符号的更多出现是如何影响最终输出的平均二进制位数的。当然，如果遇到符号 C 时就能重置 VLC 表，从而得到用码字 0 表示所有的 C 这样理想的结果就更好了，此时得到如下的输出：

< A > ,0,0, < B > ,1,1,1,0, < C > , < RESET > 0,0,0,0,0

结果表明，我们完全可以采用与字面值相同的策略，创建一个 < RESET > 词条，如下示例表所示。当解码器遇到这个词条时，它就会重置符号表并重新开始解码。编码与解码的工作原理与前面介绍的相同。

< RESET > 和 < LITERAL > 会一直在符号表中存在（像其他符号一样），但随着时间的推移，这两个符号出现得越来越少，因此其出现概率也变得越来越小。

下表就展示了 < RESET > 和 < LITERAL > 这两个词条最终成为小概率符号的情况。

符号	概率	码字
< LITERAL >	0.05	1110
< RESET >	0.05	1111
A	0.4	00
B	0.3	10
C	0.2	110

6.2.4　知道何时重置

那么，怎么知道何时需要输出重置这一词条呢？

为了做出重置的决定，需要做以下 3 件事。

- 为重置设定一个阈值，也就是说，当符号平均二进制位数（bits-per-symbol，BPS）为某个值时，放弃现有的 VLC 表并重新开始。
- 大致计算一下当前输出流的 BPS，并与设定的阈值比较。
- 计算当前已读取的输入流的熵。

当输出流的 BPS 超过设定的阈值时，例如比 BPS 大 5 个二进制位，就可以认为数据流已经发生了显著变化，应该重置概率值。

具体来说，如果一直关注输入符号的熵，我们就会发现输出流的二进制位数通常要比根据熵计算出来的大，用公式表示就是

熵 × 目前已读的符号数 < 输出流的二进制位数

这是因为受现代硬件的影响，二进制位数不可能有小数。作为替代，可以用输出流的二进制位数除以已读的符号数来得到“输出符号的平均二进制位数”，如下所示：

输出符号的平均二进制位数 = 输出流的二进制位数 / 目前已读的符号数

当比较熵与输出符号的平均二进制位数时，比较结果就会表明输出流偏离预期 BPS 的程度。

当偏离的程度大于设定的阈值时，应该重置 VLC 表，因为此时输出流已经**太过**冗余膨胀了。

阈值不是硬性规定，其取值也与数据流本身及编码器有关。每种支持此种重置的编码器，都可以根据处理数据的不同而对相应的参数调整优化。

6.2.5 实际中的应用

值得指出的是，在实际场景中没有人会使用这种简版的自适应 VLC 算法。同样的问题也存在于静态版的 VLC 算法中。相反，大多数的现代压缩工具已完全采用自适应版的哈夫曼编码器与算术编码器，它们都可以动态生成概率表，并实时更新符号对应的码字。

不过，学习这几节的内容并非是在做无用功。这些概念是动态 VLC 算法（即动态概率表、重置和字面值）的力量之源，并且大量应用于自适应哈夫曼编码与自适应算术编码中。

6.3 自适应算术编码

要将算术编码变成自适应的很容易，这主要是因为其编码步骤与概率表之间的交互很简单。只要编码器与解码器在更新概率的正确顺序上达成一致，我们就能根据需要更新概率表。

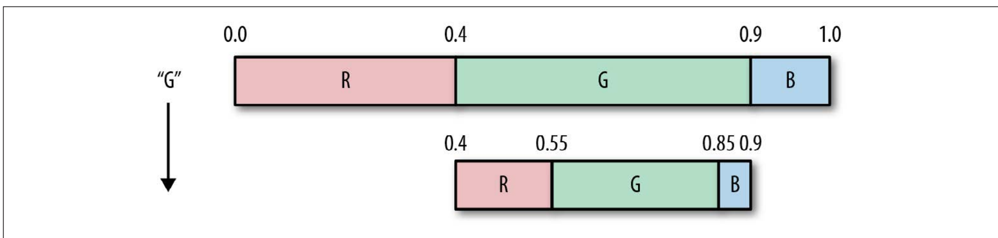
下面来看一个简单的例子，假定当前的概率表如下表所示。

符号	概率	原始区间
R	0.4	[0, 0.4)
G	0.5	[0.4, 0.9)
B	0.1	[0.9, 1.0)

(1) 读取下一个输入符号，假定是字母 G。
(2) 按 G 当前的概率对其进行编码。
(3) 根据新信息更新概率表。（由于不知道之前的输入流，因此这里假定下表所示的就是更新后的概率值。）
(4) 根据概率表重新分配区间。

符号	更新后的概率	更新后的区间
R	0.3	[0.4,0.55)
G	0.6	[0.55, 0.85)
B	0.1	[0.85, 0.9)

用图表示这一过程，则如下图所示。



解码器以相反的方式工作。给定当前的概率，找出与当前输出值对应的符号，然后更新概率表，再重新分配各符号的区间。

增加字面值与重置词条后，其工作原理仍然与自适应 VLC 相同。我们可以将这些词条指定为附加符号，并相应地调整它们的权重。

6.4 自适应哈夫曼编码

将哈夫曼编码变成自适应的不像算术编码那样简单，主要原因是哈夫曼树结构的处理比较复杂。

考虑这一问题：为了正确地输出符号的码字，需要一棵完整的哈夫曼树。最简单的想法就是，每遇到一个符号就去重新生成一棵完整的哈夫曼树。这一想法可以实现，但这样做会极大地影响算法的性能。

因此，自适应哈夫曼算法没有每次都重新生成完整的树，而是在读取和处理符号时调整现有的树。这就让情况变得稍微有些复杂，因为每读取一个符号都必须要做以下这些事情：

- 更新概率；
- 对树的大量结点变换位置并重新排序，以使它们与概率的变化同步；
- 使树的结构满足哈夫曼树的要求。

自适应哈夫曼算法的最初版本是由 Faller[7] 于 1973 年提出的，1985 年高德纳[8] 又对此算法做出重大改进，但是所有现代的版本都是建立在 Vitter[9] 于 1987 年提出的方法之上。如果你想深入研究，可以参阅脚注里提到的这些文献。

6.5 现代的选择

相比静态的方法，这些动态的改进有以下优点。

- 有生成符号码字对应表的能力，无须将符号码字对应表显式地存储在数据流中。数据流变小后，计算性能就能有所提高，但更重要的是下面两个优点。
- 有实时压缩数据的能力，无须再将整个数据集作为一个整体来处理。这让我们可以有效地处理更大的数据集，甚至都不用事先知道要处理的数据集有多大。

注 7：参见 Newton Faller 的论文 *An Adaptive System for Data Compression*，载于 *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers* (IEEE, 1973)，第 593~597 页。

注 8：参见高德纳的论文 *Dynamic Huffman Coding*，载于 *Journal of Algorithms*，1985 年第 6 期，第 163~180 页。

注 9：参见 Jeffrey S. Vitter 的论文 *Design and Analysis of Dynamic Huffman Codes*，载于 *Journal of the ACM*，1987 年 10 月，第 34 期，第 825 页。

- 有适应信息局部性的能力，即邻近的符号会对码字的长度有影响，这可以显著提高压缩率。

这三点对于现代的统计编码算法来说都很重要。随着数据量的增长，越来越多的数据需要通过互联网传输，同时，越来越多的人在移动设备上使用数据，而移动设备的存储有限，套餐流量也有限。因此，大多数的统计编码算法主要关注的是图片（WebP）和视频（WebM、H.264）文件的压缩。

这对我们来说意味着，如果处理的是少量的数据，那么简单的静态统计编码算法就可以工作得很好，并且可以让我们以较低的复杂度实现熵。如果处理的是大量的数据或者多媒体数据，而且运行时的性能很重要，那么采用自适应统计编码算法无疑是正确的选择。

第 7 章

字典转换

虽然信息论的创立是在 20 世纪 40 年代，哈夫曼编码的提出是在 20 世纪 50 年代，而互联网的出现在 20 世纪 70 年代，但是直到 20 世纪 80 年代，数据压缩才真正引起了人们的兴趣。

随着互联网的快速发展，人们分享的内容已不再局限于文字，而是开始分享比文本大得多的照片以及其他格式的数据。同时这种分享又发生在网络带宽有限、存储昂贵的时期，数据压缩因此成了缓解这些瓶颈的关键。



随着移动设备正日益成为人们访问互联网的首选，实际上我们今天还是遇到了同样的瓶颈。

虽然 VLC 一直都在发挥作用，但它与熵绑定的事实也限制了数据压缩未来的发展。因此，当大多数研究人员在尝试寻找更有效的 VLC 技术时，[1] 也有少数研究人员选择了不同的路，他们找到了使统计压缩可以更有效地预处理数据的新方法。

这种新方法通常被称为“字典转换”（dictionary transforms），它完全改变了人们对数据压缩的认知。突然间，压缩变成了一种对各种类型的数据都有用的算法。它的应用范围非常广泛，事实上今天所有的主流压缩算法（比如 GZIP 或者 7-Zip）都会在核心转换步骤中使用字典转换。下面来具体看一看。

注 1：严格来说，只 Peter Elias 一个人就提出了 30 多种 VLC 技术。

7.1 基本字典转换

统计压缩主要关注数据流中单个符号的出现概率，这一概率与其周围可能出现的符号无关。这样做虽然可以将 π 压缩为 *N* 位数字，但忽略了真实数据的基本属性：上下文及词语的组合，或者简单地说就是短语。



在其他场景下也存在着“短语”，比如音乐的规则、图像中的色彩构成或者心脏的跳动。一般来说，任何出现可以重复使用的相似内容分组的地方，都会有“短语”存在。

例如，对短语“TO BE OR NOT TO BE”，不必将每个字母都当作一个符号去编码，而将实际的英语单词当作符号去编码。这样一来，创建的符号码字对应表就会如下表所示（忽略单词间的空格）。[2]

符号	频率	码字
TO	0.33	00
BE	0.33	01
OR	0.16	10
NOT	0.16	11

这样编码后，得到的结果为 000110110011。按原来的方式对每个字母编码，最终的结果需要 104 个二进制位；而按现在这种方式对每个单词编码，最终的结果只需要 12 个二进制位。

如果考虑的对象不再是单个的符号，而是一组相邻的符号，[3] 我们就走出了统计压缩的世界，来到了**字典转换**的世界。

字典转换的工作方式也正如你期望的那样：给定源数据流，首先构建出单词字典（而不是符号字典），然后再将统计压缩应用到字典中的单词上。

字典转换并非是要去替代统计编码，相反，它只是你先应用到数据流上的一个转换，这样统计编码算法就能更有效地对其编码，如图 7-1 所示。

注 2：为什么这张表中的频率相加之和不为 1，这是因为出现了舍入误差，1/6 等于 16.666 66...%。

注 3：或者更确切地说，统计压缩接受我们扔给它处理的任何符号；而字典转换接收的是符号集，并重新定义要使用的符号以减小生成的数据流的熵。

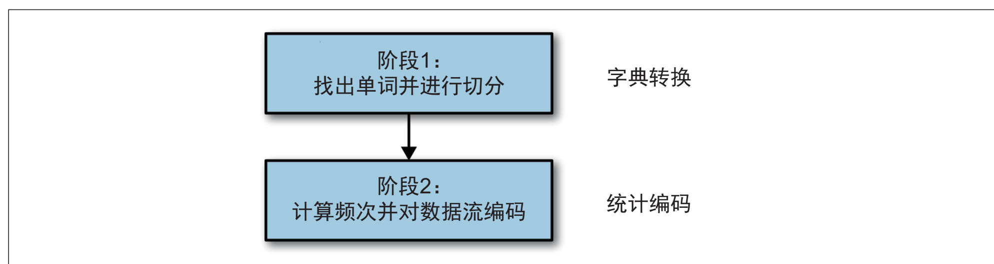


图 7-1：对源数据进行字典转换后生成的数据流，可以由统计编码算法更有效地压缩

因此，字典转换实际是一个数据流的预处理阶段，经过这样的预处理后，生成的数据集会更小，比源数据流压缩率更高。

当能识别出那些经常重复使用的长字符串，并为它们分配最短的码字时，字典转换的效率最高。

找出正确的“单词”

“什么样的单词是最佳的‘单词’”，这是个大问题。这里，最佳的“单词”指的是那些能生成最小熵的词。还有一个更大的问题是，怎样才能决定哪些词是最佳的“单词”[4]。

前面举的例子可能太简单了：通过空格就能将“单词”分出来，一眼看过去就能识别出重复出现的“单词”。

那么，下面这个字符串呢？可能会难一些吧？

TOBEORNOTTOBEORTOBEORNOT

既然没有简单的办法能将此字符串中的“单词”分出来（在不教计算机学习英语的情况下），怎样才能找出其中的“单词”呢？

可以通过被称为**分词**（tokenization）的过程找出这些“单词”，即分析一组数据并从中找出理想的“单词”。分词这一过程相当复杂，它本身也是信息论领域的一个研究分支（并有相关专利）。本书将只介绍其基础知识。

作为基础，我们先来看一看，如果根据单个符号的值也就是“字母”去分词，示例数据流会是什么样子，结果如下表所示。

注 4：这里我们使用“字母”（letter）和“单词”（word）这两个术语分别表示“单个的符号”和“多个相邻的字符”。需要说明的是，字典转换可以应用到任何类型的数据上，而不仅仅是文本。

TOBEORNOTTOBEORTOBEORNOT

字母	# 出现次数
O	8
T	5
B	3
E	3
R	3
N	2

熵=2.38

根据“字母”去分词，“O”和“T”重复出现次数最多，最终得出熵为 2.38。

得到这个结果很不错，下面尝试一下其他方法。不再用最短的符号即单个“字母”，而是用字串中重复出现的最长的子串来分词，所得结果如下表所示。

TOBEORNOTTOBEORTOBEORNOT

最长的字串	# 出现次数
TOBEORNOT	2
O	2
T	1
B	1
E	1
R	1

熵=2.5

其中最长的子串是“TOBEORNOT”，它在示例数据中出现两次。如果为它分配一个单独的码字，这样分词后字符串的熵约为 2.5，比根据字母去分词还大，因此对这样的数据来说，这不是一个好结果。

熵增加的原因是，分词之后数据中没有明显出现一个“单词”占主导地位的情况。[O,T,B,E,R,TOBEORNOT] 这些“单词”基本上是等可能出现，因此分配的二进制位数也大致相等。

也可以根据最常出现的子串去分析，这样分词的结果是 TOBEOR 和 NOT 成为了“单词”，如下所示。分词后的熵为 2.2，虽然比前面的结果好一些，但是不能给人留下深刻印象。

TOBEORNOTTOBEORTOBEORNOT

最常出现的子串	# 出现次数
TOBEOR	3
NOT	2

熵=2.2

因此，不妨再试试另一种不同的方法，通过找出长度大于 1 的最短“单词”来分词，如下表所示，TO、BE、OR 和 NOT 是切分出来的单词。分词后的熵为 1.98，这是目前得到的最好结果。

TOBEORNOTTOBEORTOBEORNOT

长度大于1的最短“单词”	# 出现次数
TO	3
BE	3
OR	3
NOT	2

熵=1.98

又回到了依靠英语“单词”来分析字符串的方法上。虽然这样做生成的熵最小，但是我们很难弄清楚怎样分析字符串才能创建出最佳大小的“单词”。

一种暴力方法是读取一组符号（如“TO”）并搜索字符串的剩余部分来确定该组符号的出现频次。如果出现频次与现有的符号表匹配得很好，那么算法就继续读取下一组符号并重复这一过程。否则，算法就会尝试读取一组不同的符号（比如“TOB”）。可惜的是，对所有真实的数据流而言，这样做不仅需要大量的内存，同时还需要花费很长的时间。因此，它不适用于任何类型的实时处理。

真相是，为了找到数据流的理想分词，我们需要有某种方法来处理现有的和那些还没有遇到的符号，并能以一种高效的方式将两者合并为尽可能长的符号集。

7.2 LZ算法

1977 年，两位研究人员 Abraham Lempel 和 Jacob Ziv 提出了几种解决“理想分词”问题的方法。这些算法根据提出的年份分别被命名为 LZ77 和 LZ78，它们在找出最佳分词方面非常高效，30 多年来还没有其他算法可以取代它们。

走近 Lempel 和 Ziv

在数据压缩领域，Lempel 和 Ziv 两人堪称“双璧”。

Jacob Ziv 大学毕业于以色列理工学院，随后于 1961 年获得了麻省理工学院信息论专业的博士学位。Ziv 对通信工程领域充满热情，之所以选择在麻省理工攻读博士学位，是因为看到该领域的顶尖学者克劳德·香农、Peter Elias 和 Bob Gallager 都齐聚在这里，他也想来到这一领域的世界研究中心。博士毕业后，Ziv 在贝尔实验室工作了一段时间，之后回到以色列理工学院成为了一名教授。

Abraham Lempel 也有类似的故事。他在以色列理工学院获得了学士、硕士和博士学位，之后成为了该校的一名教授并在这里遇到了 Ziv，从此开启了信息理论研究工作的生涯。

Lempel 和 Ziv 在信息理论领域做出了巨大的贡献，并因此在 1997 年获得了 IEEE 信息理论学会的香农奖（Claude E. Shannon Award）。

由 Lempel 和 Ziv 提出的 LZ77 和 LZ78 算法产生了一系列的衍生算法，包括 GIF 图像格式中使用的 LZW（即 Lempel-Ziv-Welch）算法，应用于 7-Zip、xz 等压缩工具的 LZM（即 Lempel-Ziv-Markov chain）算法。这些算法也同样应用于 DEFLATE 这样的压缩算法中，而 DEFLATE 又应用于 PNG 图像格式、PKZIP、GZIP 等压缩工具及 zlib 库中。

如果你有兴趣知道完整的故事，不妨看一看视频 *Compressor Head*。

7.2.1 LZ算法的工作原理

LZ 算法尝试通过在读取的字符串中寻找当前单词的匹配来分词。与读取一组符号然后向后查找它是否重复出现不同，LZ 算法向前查找当前单词是否出现过。这样做会对编码过程产生如下两个重要影响（见图 7-2）。

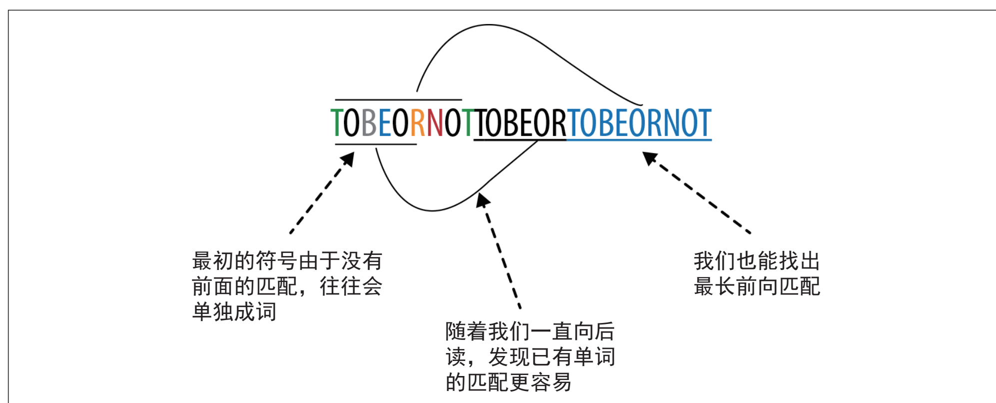


图 7-2：LZ 算法向前查找最长前向匹配单词

- 在数据流的前半部分，由于我们见过的单词很少，因此出现新单词的可能性很大；而在数据流的后半部分，由于已经有了很大的缓冲区，因此出现匹配的可能性更大。
- 向前寻找匹配可以让我们找出“最长的匹配词”。

1. 搜索缓冲区

LZ 算法的工作原理是将数据流分成如下两部分。

- 数据流的左半部分通常被称为“搜索缓冲区”（search buffer），包含的是已经读过并处理过的符号。
- 数据流的右半部分则被称为“先行缓冲区”（look ahead buffer），包含的是将要编码的符号。

因此，当前“读取”的位置就位于两个缓冲区之间，如图 7-3 所示。

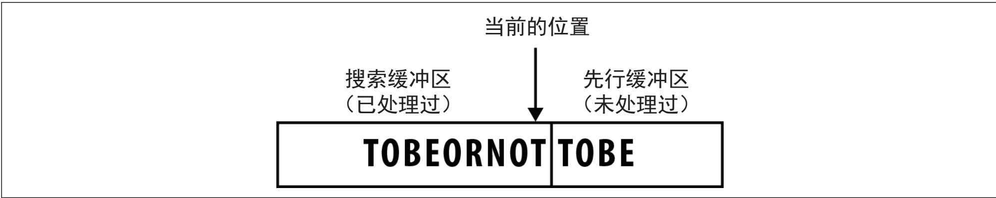


图 7-3：搜索缓冲区和先行缓冲区被当前读取的位置分开

2. 找出匹配

找出匹配其实就是搜索缓冲区与先行缓冲区之间一种有机的相互作用。

图 7-4 到图 7-9 展示了找出匹配的过程。

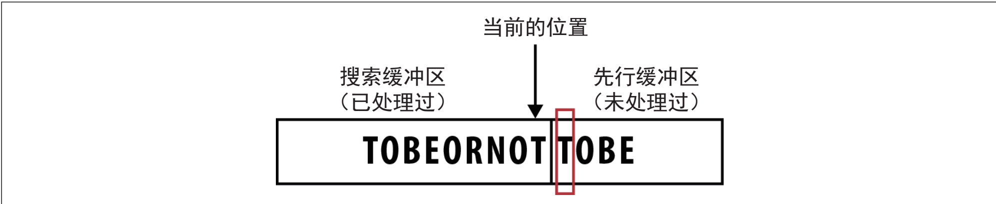


图 7-4：从当前位置读取一个符号 T

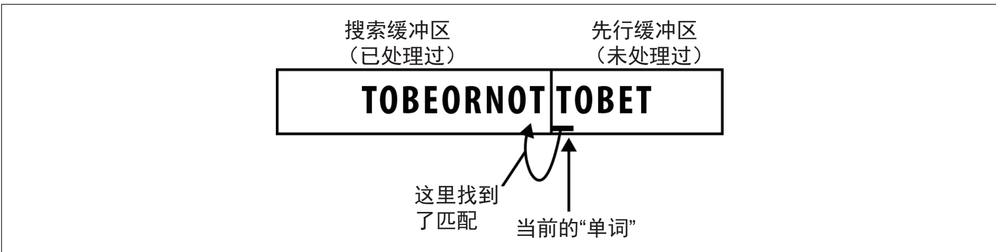


图 7-5：从搜索缓冲区往前查找，看到的第一个符号 T 就匹配上了

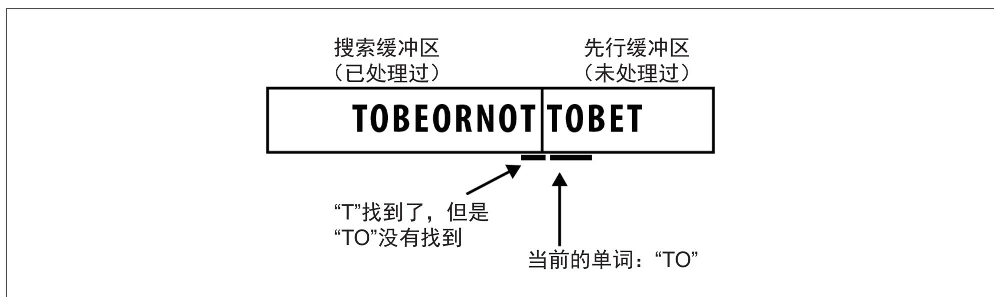


图 7-6：由于要找到的是可能的最长匹配，现在从先行缓冲区中读取第二个符号，其值为 O

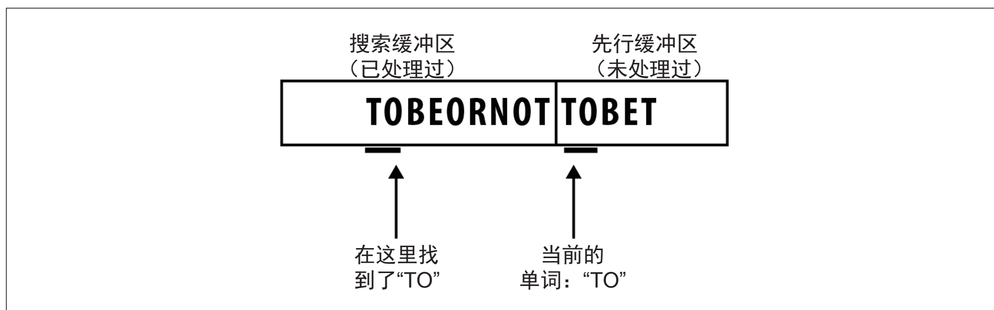


图 7-7：由于搜索缓冲区中找到的匹配 T 后没有出现 O，因此需要继续往前查找，直到最终找到 TO

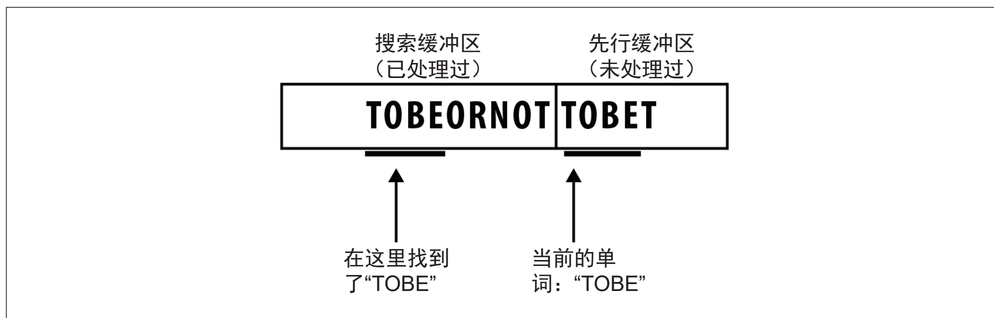


图 7-8：现在读下一个符号 B，仍能找出匹配；继续读下一个符号 E，还是能找出匹配，但当接着读下一个符号 T 时，就找不到匹配了，因此找出了这个符号序列的最长匹配



图 7-9：对找出的匹配编码（具体描述见下一节）并将“当前位置”移到先行缓冲区中最长匹配单词后，继续重复这一过程

3. “滑动窗口”

然而，在实际实现时，数据流中可能有上百万个词条，我们不可能去搜索所有处理过的符号。因为如果不对搜索缓冲区的长度加以限制，就会遇到内存及性能方面的问题，所以搜索缓冲区通常只会包含 32 KB 已经处理过的字符。因此，当移动当前位置时，搜索缓冲区的**“滑动窗口”**（sliding window）也会跟着移动，如图 7-10 所示。



图 7-10：在找出匹配并编码后，将“当前位置”移到先行缓冲区中最长匹配单词之后，此时滑动窗口也会跟着移动到新的当前位置

有了滑动窗口，查找匹配的性能要求也就有了上限。它同样也考虑到了局部性原理，即在给定的数据集中相关的数据很可能分布在相似的局部区域。



一般来说，搜索缓冲区滑动窗口的长度大概为几万个字节，而先行缓冲区的长度则只有几十个字节。

4. 用记号标记匹配

当匹配最终确定下来，编码器就会生成一个固定长度的记号并将它写入输出流。该记号主要由两部分组成：偏移量和长度。[5]

- **偏移量**

  该值表示的是搜索缓冲区中匹配单词的起始位置，从当前位置向前数。在前面举的例子中，匹配的字符串需要从当前位置往前数 9 个字符。

- **长度值**

  该值表示的是匹配单词的长度。在本例中，匹配单词的长度为 4（即包含 4 个符号）。

具体到本例中，找到的匹配位于当前位置 9 个符号前，且其长度为 4，因此将二元组 [9,4] 写入输出流，如图 7-11 所示。

注 5：值得注意的是，在最初提出 LZ77 和 LZ78 算法的论文中，该记号其实是一个三元组，第三个值是先行缓冲区中的下一个符号，它在解码时有助于符号处理和恢复。不过，现代大多数的 LZ 系列算法已不需要第三个值了，因此通常我们会忽略它。

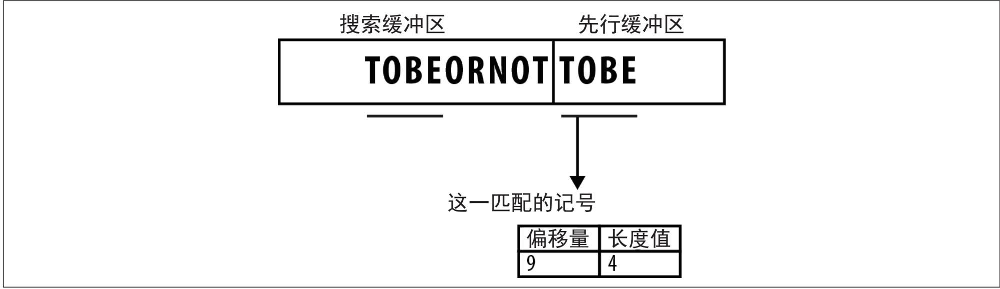


图 7-11：位于先行缓冲区中与搜索缓冲区中符号匹配上的单词，它与匹配符号的偏移量与长度值编码

解码器会以非常简单的方法逆转换这些值：

(1) 读取下一个词条；

(2) 以当前位置为起点，在搜索缓冲区中往前数偏移量个符号；

(3) 抓取长度值个符号并添加到数据流后面。

5. 没有找到匹配时

在一些情况下，无法在搜索缓冲区中找到先行缓冲区中出现符号的匹配。这种情况下，需要输出一些信息来表示这个新出现的单词，这样解码器才能正确地还原它。

因此，需要对输出的记号进行修改，表明输出的是字面值，这样解码器就能读取并恢复源数据流。不过，怎样构造该记号完全取决于具体的 LZ 算法实现。一种最基本的做法是，将修改后的记号表示为三部分，前两部分与前面介绍的相同，还是偏移量和长度值，只不过取值都为 0，即 [0,0], 最后一部分则是符号的字面值，如图 7-12 所示。

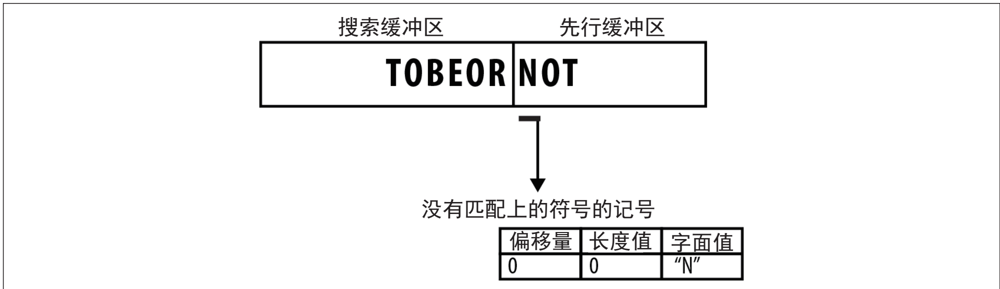


图 7-12：没有匹配上的符号的记号与前面介绍的不同，为了方便解码器的读取，记号中通常会包含符号的字面值

7.2.2 编码

以字符串“TOBEORNOTTOBE”为例，走一遍编码的过程（同时也请参见随后的表格）。

(1) 开始的 4 个符号在搜索缓冲区找不到匹配，直接输出字面值记号。

(2) 当在先行缓冲区中读到第二个“O”时，我们在搜索缓冲区找到一个单个值的匹配，因此输出记号（3,1）。
(3) 继续这一过程，发现找不到匹配或者仅找到了单个值的匹配。
(4) 当读到字符串的结尾时，情况变得有意思起来，我们发现在搜索缓冲区的最开始找到了 TOBE 的匹配，其位置为 9，长度为 4。

下表展示了整个编码过程。

搜索缓冲区	先行缓冲区	输出
	TOBEORNOTTOBE	0,0,T
T	OBEORNOTTOBE	0,0,O
TO	BEORNOTTOBE	0,0,B
TOB	EORNOTTOBE	0,0,E
TOBE	ORNOTTOBE	3,1
TOBEO	RNOTTOBE	0,0,R
TOBEOR	NOTTOBE	0,0,N
TOBEORN	OTTOBE	3,1
TOBEORNO	TTOBE	8,1
TOBEORNOT	TOBE	9,4
		< eos >

7.2.3 解码

整个解码过程全靠读取输入的记号。

- 当解码器读到字面值记号时，就将该值输出到恢复缓冲区；
- 当解码器读到“匹配”的记号时，会从当前的位置往前数偏移量个符号，以此为起点将长度值个符号添加到恢复缓冲区的结尾。具体过程如下表所示。

输入的记号	恢复缓冲区
0,0,T	
0,0,O	T
0,0,B	TO
0,0,E	TOB
3,1	TOBE
0,0,R	TOBEO
0,0,N	TOBEOR
3,1	TOBEORN
8,1	TOBEORNO
9,4	TOBEORNOT
< eos >	TOBEORNOTTOBE

看明白了吗？还是很简单的吧？

7.2.4　压缩LZ算法的输出

很容易看出，经过 LZ 变换后所生成的编码后的数据流比源数据流小。（在本书中，我们认为任何用 2 个符号的记号去替换 12 个符号的单词的机会都是一种胜利。[6]）对那些有很多重复单词的数据流来说，你可以用小得多的记号对它进行编码，这就是 LZ 算法吸引人的地方。

然而还不止如此，LZ 算法真正吸引人的地方还在于它可以和统计编码结合使用。可以将记号中的偏移量、长度值以及字面值分开后，再按照类型合并，组成单独的偏移量集、长度值集和字面值集，然后再对这些数据集进行统计压缩。

例如，可以将前面例子输出的记号集 [0,0,T][0,0,O][0,0,B][0,0,E][3,1][0,0,R][0,0,N][3,1][8,1][9,4] 分成以下 3 个数据集。

偏移量集　0,0,0,0,3,0,0,3,8,9
长度值集　0,0,0,0,1,0,0,1,1,4
字面值集　T,O,B,E,R,N

这三个数据集的性质不同，相应的处理方法也不同。

1. 偏移量集

首先，我们知道偏移量永远都是在 [0,*X*] 这个范围之内，其中 *X* 是搜索缓冲区的长度值。在最坏的情况下，偏移量也可以用 lb(*X*) 位来编码，这就允许我们对滑动窗口内的任意字节进行索引。

不幸的是，偏移量的取值很分散，因此对于大的数据流来说，偏移量集中不会存在很多重复的内容。不过即使这样，进行统计编码后仍然可能会产生较好的结果。例如，举的例子中偏移量集的熵为 1.57，但要知道，随着数据集变大，情况会变坏。最坏的情况就是在搜索缓冲区中到处都存在匹配，这样一来，偏移量集中的每个值都不相同。

2. 长度值

长度值也存在着类似的问题。它们可以取任何值，因此唯一的希望就是利用重复符号通过统计编码算法来进一步压缩数据。长度值的分布取决于输入流的内容以及语言的类型。例如，如果要编码的是一本用英语写的书，那么长度值中出现最多的就会是 2、3 或者 4。具体到前面举的例子，长度值的熵为 1.30，也就是说约需要 13 个二进制位来对这些长度数据进行编码。

注 6：作为作者，这是我们写的书，因此我们总是在赢。

3. 字面值集

对我们举的这个小例子来说，与偏移量集和长度值相比，字面值集好像也没有什么更好的压缩方法。不过，随着输入流增大，由于可能会存在重复的字面值（这是因为有滑动窗口），因此字面值流的熵也会略微变小。当然，这种情况是否会发生还取决于搜索缓冲区的大小。例如，如果输入流中存在两个符号 B，但是被 32 000 个其他符号隔开，由于隔得太远，这两个符号 B 也就匹配不上。因此，字面值流中会出现两个符号 B。

7.2.5 LZ算法的变体

LZ 算法很强大，让人印象深刻，但同样令人印象深刻的还有过去 40 多年来出现的该算法的变体数量（见图 7-13）。每一种变体都是根据特殊的需要、性能要求的不同或者用例的不同，对 LZ77 基本算法进行了一些小调整。下面选择一些重要的加以介绍，剩下的算法有待你自己去探索。

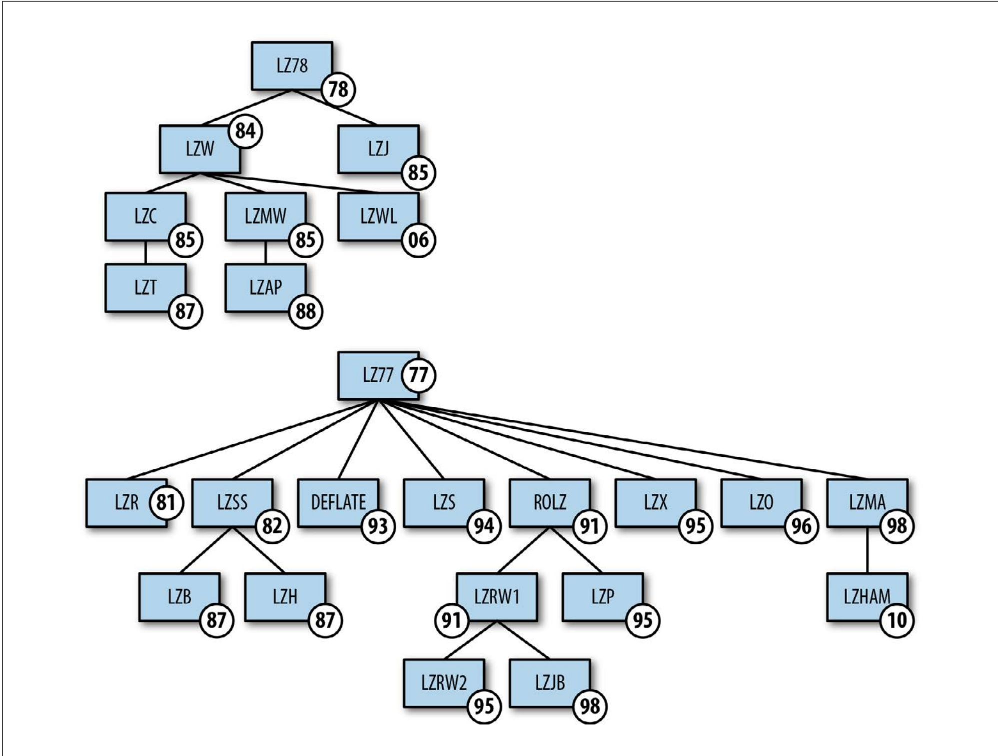


图 7-13：LZ77 与 LZ78 系列算法的谱系图，图中展示了这两个算法的各种变体以及它们的提出年份

1. LZ77

LZ77 基本算法（有时也被称为 LZ1 算法）的工作原理，与前面介绍的大致相同。唯一的区别是，它会将先行缓冲区中下一个符号的字面值作为第三个值输出。

2. LZSS

LZ77 与 LZSS 的主要差别是：在 LZ77 算法中，字典引用可能会比其替换的字符串还长；而在 LZSS 中，如果被替换的字符串长度值小于“收支平衡”点，那么这样的引用就会被忽略。此外，LZSS 还会用一个标志位来区分后面的数据是字面值还是偏移量 – 长度值二元组这样的引用。

很多流行的压缩工具比如 PKZip、ARJ、RAR、ZOO 和 LHarc 使用 LZSS 算法，并将其作为主要的压缩算法。值得一提的是，Game Boy Advance 游戏机的 BIOS 就自带解码改进后的 LZSS 格式补丁等功能。

3. LZ78 或 LZ2

LZ 算法系列的核心算法最早发表于 1977~1978 年，这两种算法有时也被称为 LZ1 算法和 LZ2 算法。LZ78 算法的工作原理与前面描述的基本相同，不过它不用距搜索缓冲区结尾的偏移量来指示匹配的位置，而是根据输入流创建字典然后再引用。

4. LZW 算法

LZW（Lempel-Ziv-Welch）算法是由 Terry Welch 于 1984 年提出的，它采用了 LZ78 算法的思想，其工作原理如下。

(1) LZW 算法将字典初始化为包含所有可能的输入字符，如果用到了清空和停止符号（clear and stop codes)，那么这两个符号也包括在其中。
(2) 该算法扫描输入字符串以寻找更长的连续子串，直到它发现该子串在字典中不存在。
(3) 当发现这样的子串时，去掉它的最后一个符号（这样它就变成当前字典中最长的子串），然后从字典中找出其索引并输出。
(4) 将该子串（此时包括最后一个符号）加入字典作为新的词条。
(5) 将该子串的最后一个符号作为起点，重新扫描下一个子串。

用这种方法，连续更长的子串就会作为新的词条加入字典，同时也让后续字符串编码为单值输出成为可能。该算法最适用于那些连续出现重复的数据，因为在数据的初始部分，基本看不到什么压缩，但是随着数据的增多，压缩率逐渐趋于最大值。

LZW 算法成为首个在计算机中广泛采用的通用数据压缩方法。公共领域程序“compress”也采用了 LZW 算法，并在 1986 年前后就基本成为 UNIX 系统的标准应用程序。此后，“compress”就从很多 UNIX 分发中消失了，一方面是因为它侵犯了 LZW 的专利权，另一方面是因为 GZIP 的压缩率更高（GZIP 使用的是基于 LZ77 的 DEFLATE 算法）。

7.3　尽可能多地收集数据

这里我们想指出的是，潜在的输入数据集的量是很大的，而每个数据集都可能以一种特殊的方式去响应一种算法。对数据集越了解，你就越能从中选择出最适合的 LZ 变换。

第 8 章

上下文数据转换

在开始本章内容前，我们先花一些时间回顾一下前面的内容。

统计编码算法工作时会为每个符号分配一个长度可变的码字，压缩主要来自于为越频繁出现的符号分配越短的码字。而字典转换的分词过程会识别出数据集中最长且最频繁出现的那些符号。实际上，只有分词过程找出了那些最适合的符号，编码的效率才能提高。从技术上来说，我们完全可以通过分词过程识别出那些最适合编码的符号，然后再将所得结果交给统计编码算法处理以得到压缩的效果。然而，LZ 算法的真正威力体现在我们没有那样去做，而是用熵值较低的记号二元组来表示匹配的信息，然后再去压缩这些记号二元组。

除了字典变换之外，还有一整套其他的变换都是按照同样的原理工作的：给定一组相邻的符号集，对它们进行某种方式的变换使其更容易压缩。我们通常称这样的变换为“上下文变换”（contextual transform），因为在思考数据的理想编码方式时，这些方法考虑到了邻近符号的影响。

这些变换的目标是一致的，即通过对这些信息进行某种方式的变换，使统计编码算法对其进行压缩时更高效。

变换数据的方法有很多种，但其中有 3 种对现代的数据压缩来说最为重要，即**行程编码**（run-length encoding，RLE）、**增量编码**（delta coding）和**伯罗斯 – 惠勒变换**（Burrows-Wheeler transform，BWT）。

下面分别介绍这 3 种方法。

8.1 RLE

RLE 是过去 40 多年来看似很简单、实则很高效的编码技术之一，适用于各种类型的数据。RLE 主要针对的是连续出现的相同符号聚类的现象，它会用包含符号值及其重复出现次数的元组，来替换某个符号一段连续的“行程”（run）。例如，如图 8-1 所示，字符串 AAAABBBBBBBBCCCCCCCC 就可以编码为 [A,4][B,8][C,8]。[1]

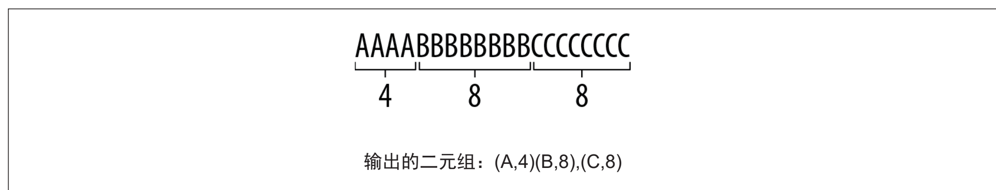


图 8-1：RLE 在识别符号流中相同符号的行程（连续出现的相同符号），它会将这样的数据流转换为包含符号值及其长度值的二元组

从概念层面来说，RLE 就是这么简单，没有什么特别之处。编码工作就是找到一个符号并向前扫描看看其行程有多长。

解码工作则相反，给定某个符号值及其长度值的二元组，只需要将正确个数的符号添加到输出流之后就行了。

8.1.1 处理短行程问题

然而，并非所有的数据都像前面举的例子那样规律一致。根据 RLE 的算法，字符串 AAAABCCCC 会被编码为 [A,4][B,1][C,4]。由于字符串中间只有 1 个 B，因此编码后 B 由一个字符变成字符及其长度值的二元组，反而变大了，如图 8-2 所示。



图 8-2：短行程是 RLE 作为一种算法面临的大问题。存储短行程的开销极大地影响了数据压缩后的大小，本图就是很好的例子，很多并非重复出现的符号让输出变大了很多

注 1：根据现有的计算机科学课程及维基百科，RLE 通常不会用上面这样成对的记号来表示，而是将 A4B1C4 这样的符号值及其长度放在一起来表示。然而，由于符号与数值会交替出现，因此实际上没有人会使用这种形式的 RLE，介绍这种形式纯属浪费时间。很抱歉，让你分散注意力来看这个脚注。

如果幸运的话，那么存储这些单独出现的符号的开销，很容易就被存储那些长行程的符号所节省的空间抵消了。

如果是其他情形，就需要辨别出那些用 RLE 算法编码后反而变大的符号，对这类符号，可能需要将它们单独留在数据流中。例如，可以只对那些行程长度大于 2 的符号编码。

有了这样的假定，字符串 AAAABCCCC 就可以编码为 [A,4]B[C,4]。因此，如果很多字符不是连续重复出现，就没必要使用计数器。这样处理带来的问题是，解码时很可能会出现歧义。如果将编码后的数据流转换为二进制，那么最终得到的结果为 1000001100100001010000111002[2]，这样就无法分清字符 B 从哪里结束以及字符 C 从哪里开始。一般来说，数据流中交错出现字面值是会出问题的。[3]

因此，需要有一种方法来区分哪些字符编码后是二元组哪些字符不是。通常采用的方法是，在数据集中增加一个二进制位流，来表示某个给定的符号流中各个符号是否连续重复出现（见图 8-3）。因此，要在 100000110010000101000011100 之前加上二进制位流 101，它表示第一个符号连续重复出现，第二个符号没有连续重复出现，第三个符号又连续重复出现。这样，通过为每个行程（符号）增加 1 个二进制位的标志位，就节省了存储短行程所需的额外开销。

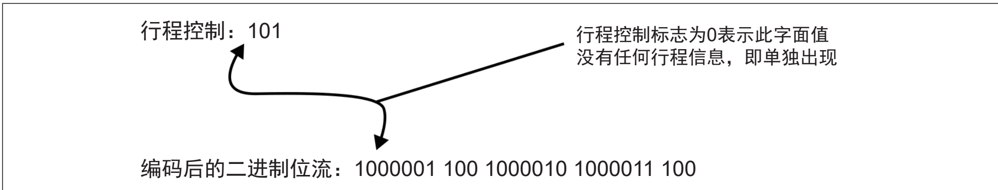


图 8-3：有了另外的二进制位流表示哪些字面值是行程，就能正确地解码了。在这个例子中，第二个字面值对应的行程控制为 0，说明不用继续读后面表示行程长度的 3 个二进制位数据



重要提示

RLE 算法最适用于大多数符号都连续重复出现的数据集。如果要处理的数据集没有这样的性质，那么 RLE 算法并不适用。你可能需要继续学习本书后面关于 MTF 或增量编码的内容。

8.1.2　压缩

压缩 RLE 算法输出的结果需要一些技巧。首先，需要将数据集分成两部分：字面值流和行程长度流。（还记得加到最前面的二进制位流吗？它会告诉我们解码时需要从哪个流中读

注 2：如果你认真观察，就会发现每个字面值都是 7 位的 ASCII 码，因此整个二进制串的正确划分为 1000001|100|1000010|1000011|100。

注 3：要使用自适应统计编码和字典转换算法，这就是必须要解决的问题，可以说如果在数值流中有字面值交错出现，那就是自找麻烦。

取数据。）对字面值流，可以根据自己的意愿选择一种编码器[4]，剩下的行程长度流才是压缩问题的真正所在。因此，下面将使用试错这种非常现实的方法，为行程长度流找到合适的编码。[5]

给定行程长度流 [4,3,1,1,1,3,1,2,1,2,1]，可以用 2 个二进制位对每个长度值进行二进制编码，所有这些值共需要 22 个二进制位。不过，如果遇到其中某个长度值特别大的情况，例如 [256,3,1,1,1,3,1,2,1,2,1]，这种方法就不适用了。这种情况下，每个长度值编码时所需的二进制位数都与集合中最大值的位数相等，也就是说编码每个值都需要 8 个二进制位，总共需要 88 个二进制位。这样的结果很不理想。

因此，不妨试试静态的 VLC。因为依据最简单的模型，我们可以为每个行程长度值都分配不同的二进制位数，所以如果长度流为 [4,3,1,1,1,3,1,2,1,2,1]，使用**一元编码**（unary encoding）得到的结果为 [1110,110,0,0,0,110,0,10,0,10,0]，共 20 个二进制位。可以看出，这种方法并不适用，它将最短码字分配给了最小的值（假定它在数据流中最频繁出现）。而 RLE 的长度则恰好相反，我们想将最短码字分配给**最大**的值（因为它表示的是最长的行程）。

如果使用统计编码算法（比如哈夫曼编码或算术编码），结果可能会更好。可以通过计算集合的熵来了解这些值对整个数据集的影响，该集合的熵为 1.69，也就是说编码后的大小约为 19 个二进制位（节省了大约 13% 的空间）。当然，如果使用的是自适应统计编码算法，熵还要再低一些，因为只要有局部性存在，它都会加以利用。

不羁的书虫

RLE 算法最初是在 1966 年由 Solomon W. Golomb 在一篇论文[6]中提出的，也是在这篇论文中，他第一次使用一种形式新颖、内容丰富的詹姆斯·邦德式比喻来描述现在著名的 Golomb 编码：

> 特工 00111 号再次回到了赌场，玩起了赌博游戏，与此同时人类的命运却悬而未决。

这篇经同行评议的基础性论文随后用玩轮盘赌桌游戏这一比喻来描述符号出现的可能性，其中甚至包含向酒保点头对整个赌局的影响。这是对那些真正醉心于数据压缩研究的人们的一个生动刻画。在这一领域内，人们很容易只关注算法而忽略背后涉及的那些人，但事实是他们都是书虫。数学书虫大多是沉闷、傻气且有些古怪的，他们热爱数字并能找到幽默的方式讨论复杂的问题。

通常认为 RLE 是单字符上下文模型，也就是说，对任何给定的符号，在编码时我们都只考

注 4：应用 LZ 算法很有趣，因为这样就可以看到符号行程重复的频率。

注 5：即便是那种有多种算法的压缩工具也可能以这种方法进行选择。

注 6：参见 S. W. Golomb 的论文 *Run-length Encodings*，载于 *IEEE Trans. Information Theory*，1966 年 7 月，IT-12 卷，第 399~401 页。

虑它的前一个符号，如果这两个符号是相同的，那么行程继续；如果不相同，那么当前行程终止。虽然 RLE 在现代压缩工具中用得并不多，但还是有人在研究更高效的 RLE。例如，最近就出现了一种新的 RLE 压缩工具 TurboRLE，号称是速度最快、效率最高的 RLE 编码器。



有时将 RLE 中的长度当成一种增量编码值是有帮助的。如果我们从绝对值的角度理解每个行程的开始，那么长度值表示的是数据流中符号变化之间的距离。

8.2 增量编码

虽然前面的章节提到了增量编码，但接下来会对其进行更深入的研究。从压缩角度来说，数值型数据算是最令人讨厌的数据类型之一，这是因为大多数时候，我们找不到可以利用的统计信息。而数值型数据到处可见，比如 GPS 的坐标信息，搜索引擎的倒排索引信息以及返回的用户 ID，这些都是数值型数据。我们来看下面这组很难处理的数据：

[51,12,8,221,0,0,12,18,9,255,0,18,64]

从熵的角度来说，我们能做的十分有限：这组数据中只有少数几个值是重复的，其余的熵值比较高。一般来说，存储这组数据中的每个值需要用 8 个二进制位的空间。幸运的是，有方法可以将这组数转换为熵值更低的另一组数。

增量编码，其实就是将一组数据转换为各个相邻数据之间的相对差值（即增量）的过程。它背后的思想是，给定一组数据，相关的或相似的数据往往会集中在一起。如果这样，有了两个相邻值之间的差，就可以用其中一个值以及该差值来表示另外一个值。一般来说，我们会用当前值减去前一个值，然后将差值写入输出流。

增量编码可以说是现代计算技术中最重要的算法之一。在数值型数据这样普遍而其熵值又如此偏高的情况下，增量编码提供了一种不依靠统计的转换。事实上，它依靠的是相邻性，所以最适用于处理时间序列数据（比如每 10 秒检测一次温度的传感器所产生的数据），以及音频和图像数据这类多媒体数据，因为这类数据中邻近的数据之间存在着时间上的关联。

以下面这组数为例：

[1,3,6,8,10]

从第二个数开始，用当前数减去前一个数，得到了增量编码之后的数据：

[1,3–1,6–3,8–6,10–8] → [1,2,3,2,2]

如果存储源数据，那么每个数需要 4 个二进制位的空间（因为 lb(10) = 4）。增量编码后，每个数只需要 2 个二进制位的存储空间，结果是所有这些数共需要 10 个二进制位的空间，而非 20 个二进制位。

要找回源数据，按照相反的步骤操作就行了，将当前数与前一个数相加就行了。

从编码后的数据集 [1,2,3,2,2] 开始，按照顺序，将当前数与前一个数之和作为新的当前数，得到了原始数据：

[1,1 + 2,3 + 3,6 + 2,8 + 2] → [1,3,6,8,10]

一般来说，增量编码的目的就是缩小数据集的变化范围。更确切地说，是为了减少表示数据集中的每个值所需要的二进制位数。这也就意味着，当相邻数值之间的差相对较小时，增量编码最有效。如果差值变大，情况就会变糟。

我们来看以下数据集：

[1,2,10,256]

增量编码之后，得到了如下的结果：

[1,2–1,10–2,256–10] → [1,1,8,246]

可以看到，增量编码后数值的变化范围并没有变小，还是平均需要 lb(maxVal) 个二进制位才能对整个数据集编码。

但这不是最坏的情况，我们来看以下数据：

[1,3,10,8,6]

增量编码后，你可能会想哭：

[1,3–1,10–3,8–10,6–8] → [1,2,7,–2,–2]

在这组数中，有些数要比前一个数小，因此转换后结果中有负数。其中最大的正数为 7，因此存储这些正数平均需要 LOG2(7) = 3 个二进制位。不幸的是，现在还需要表示负数，这就意味着还需要额外的 1 个二进制位来表示正负，因此表示每个数都需要 4 个二进制位。

这样的情况很常见，也正好是增量编码的弱项。遇到这样的数据，增量编码的效率自然不会高。不过，也不用灰心，无论需要处理的是什么样的数据，我们还是有不少方法可以改进增量编码这一算法，让它变得更加健壮。

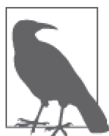
下面来看一些简单的改进方法。

8.2.1 XOR增量编码

减法增量编码算法的问题是，结果中可能会出现负数，进而产生各种问题。负数不仅在存储的时候需要额外的二进制位，此外还可能会增大数据的变化范围，例如以下数据：

[1,3,10,8,6] → [1,3–1,10–3,8–10,6–8] → [1,2,7,–2,–2]

可以通过使用按位异或运算（bitwise exclusive OR，XOR）代替减法运算来解决这一问题。



XOR

XOR 会独立地对每个二进制位进行操作。XOR 是一种逻辑运算，仅当两个输入不相同时（即其中一个为真，另一个为假时）结果为真。

例如：

```
      0101( 十进制为 5)
  XOR 0011( 十进制为 3)
    = 0110( 十进制为 6)
```

注意，与 1 XOR 相当于对该位取反。

又如：

```
      0101( 十进制为 5)
  XOR 1111( 十进制为 3)
    = 1010( 十进制为 10)
```

这里注意，任何值与自己 XOR 其结果总是 0。

在很久以前，那时寄存器还需要手工去取反和清零，这些的确都是基本知识。

XOR 完全绕开了负数出现的问题，因为整数之间的 XOR 根本不可能产生负数。

再次以 [1,3,10,8,6] 这组数据为例，由于

$1 \oplus 1 = 0$

$3 \oplus 1 = 11 \oplus 01 = 10 = 2$

$10 \oplus 3 = 1010 \oplus 0011 = 1001 = 9$

$8 \oplus 10 = 1000 \oplus 1010 = 0010 = 2$

$6 \oplus 8 = 0110 \oplus 1000 = 1110 = 14$

因此，XOR 增量编码生成的结果为 [1,2,9,2,14]。

虽然这同样没能缩小数值的变化范围，因为存储每个值还是需要 4 个二进制位的空间，但它**的确**保证了无论数值之间的相互顺序是怎样的，编码后的每个值都是正的。

8.2.2 参照系增量编码

考虑下面这组数：

[107,108,110,115,120,125,132,132,131,135]

将这 10 个数中的每个数都存储为 8 个二进制位的整数，共需要 80 个二进制位的空间，不过这看起来有些浪费，因为所有小于 107 的数都只是在填充空白。我们预留了足够的二进制位来表示这些数，但是数据集中没有这些数。

参照系方法通过让其他数减去最小的数来解决上面这个问题。在本例中，所有的数都处于 107~135。因此，与其对原始序列进行编码，不如先让每个数都减去 107，然后再对所得差值进行增量编码，即对 [0,1,3,8,13,18,25,25,24,28] 进行编码。

这样一来就可以用不超过 5 个二进制位的空间对每个差值进行编码了。

当然，还是需要用 8 个二进制位来存储最小值 107，此外需要至少 3 个二进制位来记录每个值只使用了 5 个二进制位的空间。即使如此，需要的总二进制位数也只是 56（$8 + 3 + 9 \times 5$），比原来的 80 个二进制位少很多。

“参照系”（frame of reference，FOR）中那个“参照数”（frame）的选取，与将转换恰当地应用到数据集上有关，因此需要将数据集细分为更小的数据组。

例如，可以将前面那组数拆分成以下两组数：

[107,108,110,115,120] 和 [125,132,132,131,135]

使用参照系增量编码处理之后，得到：

[107,0,1,3,8,13] 和 [125,0,7,7,6,10]

为数据集确定了参照数后，数值的变化范围缩小了，因此表示每个值所需要的二进制位数也变小了。[7]

遗憾的是，离群值还是可能造成很多问题。

“参照数”到底是什么

FOR 最初的设计目的是，尽可能地将更多数值匹配到单个整数的空间之内（通常是 32 位或者 128 位的整数）。这样做有以下两个原因。

- 它使数值在运行时更容易处理（因为计算机处理经过字节对齐，是 2 的幂的那些数值会更容易），同时还可以将它当作一种漂亮的内存压缩表示。

注 7：当然，最主要的问题还是如何确定最理想的参照数。目前，大多数实现使用的是 32 位到 128 位这样的窗口，因为这个空间刚好能表示一个整数。

- 它提供了一种非常简单的压缩方法。将 10 个整数压缩到 32 个二进制位的空间内，这样的压缩效果可以说很好了，其结果是产生了一种性能很强的方法，可以在一秒内解码数十亿个整数值，代价则是那些没有充分利用空间的整数需要额外的开销。

8.2.3 修正的参照系增量编码

继续考虑以下数据[8]：

[1,2,10,256]

增量编码后，得到：

[1,2–1,10–2,256–10] = [1,1,8,246]

这组数据中的离群值基本上破坏了对其余数据的压缩。

为了缓解这一问题，Zukowski 等人[9]提出了一种补救的方法，称为修正的参照系增量编码（Patched Frame of Reference Delta Coding，PFOR）。[10]

它的工作原理如下步骤所示。

(1) 选择一个位宽度 *b*。
(2) 遍历数据并用 *b* 位对每个值编码。
(3) 当遇到的值所需的编码位数大于 *b* 时，将这样的离群值存储在单独的位置。

PFOR 的神奇之处就在于其对离群值的处理。

(1) 考虑前面的例子增量编码后得到的结果，即 [1,1,8,246]。
(2) 用最简单的形式将这组数据分成两部分，即编码需要的二进制位数不大于 *b* 的以及大于 *b* 的。当 $b = 4$ 时，得到 [1,1,8][246] 这两组数据。
(3) 用 4 位对第一组中的每个数编码，用 8 位对第二组中的离群值编码。
(4) 为了让离群值能合并到源数据列中，需要知道它的位置信息，因此得到的结果为 [1,1,8][246][3]。

在解码时，我们需要取出离群值并将它们放回源数据流，然后再进行增量编码的逆操作。

注 8：我们知道，一进入这一章，你就思考过这组数据了。因此这一次，不妨**更认真地**思考一下。

注 9：参见 M. Zukowski, S. Heman, N. Nes 和 P. Boncz 的论文 *Super-Scalar RAM-CPU Cache Compression*，载于 *Proceedings of the 22nd International Conference on Data Engineering, ICDE '06, IEEE Computer Society: Washington, DC, USA*，2006 年，第 59~71 页，doi:10.1109/ICDE.2006.150。

注 10：值得指出的是，PFOR 是一种独立于增量编码的技术，完全可以单独地应用到数据集上，这也是为什么当与增量编码一起使用时，可以看到这一算法有时会写作 PFD、PFor 或者 PForDelta。我们喜欢这样使用它，因此将它安排在这一节，这是当作者的一个好处。

当然，说到这里还是有两个主要问题没有解决，即如何选择 *b* 的值以及怎样处理离群值？

1. 选择合适的 *b* 值

我们的目标很明确，就是选择一个合适的 *b* 值，使大多数值在编码时需要的位数不超过 *b*，并且可以通过它识别出那些离群值。

通常可以逐步做到这一点。可以从 $b = 1$ 开始试验，看看数据集中有多少数小于 2^1，如果数据集中 90% 的数小于 2，那么就将 *b* 设置为 1。否则，令 *b* 的值为 2，再看看数据集中有多少数小于 2^2。如有必要，可以重复这一过程，比如令 *b* 的值为 3 并看看有多少值小于 2^3，直到数据集中 90% 的值满足条件。

2. 怎样处理离群值

PFOR 的有趣之处在于，除了得到修改后的增量信息外，我们还得到了第二个数据集，它表示的是那些离群值。第二个数据集中的值变化范围很大，通常很难直接压缩。根据原始论文 *Super-Scalar RAM-CPU Cache Compression* 的叙述，PFOR 没有将整个的离群值原样扔进一个新的列表中，而是将最低的“b”位留在源数据流中，并将剩下的部分存储在离群值列表中。

例如，在下面这组数中，除了一个数以外，其余数最高的三位是 0：

101**0010**
000**1010**
000**1100**
000**1011**

处理这组数时，不用将 101,000,000,000 这些值都存储在离群值列表中，而是一眼就能看出只有第一个值才是需要关注的离群值。结果是得到了如下的三组新数据：

第一组就是原始数据的最低 4 位，即

[0010,1010,1100,1011]

紧接着第二组则是离群值所在的位置，即

[0]

第三组则是这些位置上离群值的实际异常位，即

[101]

这样处理后，离群值组的变化范围缩小了很多，也有可能更容易压缩。

8.2.4 压缩增量编码后的数据

注意，到目前为止实际上还没有进行任何压缩，只是对数据进行了某种方式的转换，使它变得更容易压缩。[11]

如果增量编码能做到以下两点，那么我们就可以认为它生成的数据更容易压缩：

- 将数据集中的最大值变小，因此缩小了数值的变化范围；
- 生成了许多重复值，可以让统计压缩的效率更高。

其中第二点更重要，因为这样的数据更适合用统计压缩工具压缩。然而，如果增量编码没有生成便于统计压缩的变量数据，那么我们就需要利用好第一点，朝着减小整个数据集的整体 LOG2 这一方向努力。

一般来说，将增量编码后的结果交给统计编码算法处理，会产生良好的压缩效果。

8.2.5 那么它对文本有效吗

可能不会那么有效。我的意思是，它可以工作，但是由于英语文本中使用最多的是还是字母表中两头的字母，因此得到的数据中会出现很多正负数交替的情况。另外，对于英语文本，像 LZ 这样的算法可能会做得更好。

8.3 MTF

上下文数据转换背后的基本思想是，数据的排列次序中包含着一些有助于编码未来符号的信息，前移编码（move-to-front coding，MTF）利用的也是这样的信息。然而，与 RLE 和字典编码器只考虑直接相邻的符号不同，MTF 考虑更多的是在较短的窗口内某个特定符号的出现次数。

MTF 反映了如下的预期：如果一个符号在输入流中出现了，那么它很有可能会出现多次，或者至少短时间内会成为常见的符号。MTF 是局部自适应的，因为它会根据输入流中局部区域符号的出现频次进行调整。如果输入流满足了这样的预期，换句话说，如果输入流中出现了相同符号集中的情况（即输入流中符号的局部频次会出现显著的变化），那么 MTF 会产生好的结果。

图 8-4 展示了 MTF 的完整过程，它是通过另外管理一组数据来工作的，其中包含的是数据集中所有不同的值，我们称为 SortedArray。当从数据流中读取一个值时，我们会找出该值在 SortedArray 中的索引并将此索引值输出，然后更新 SortedArray，将该值移到**最前面**，即让其索引变为 0。

注 11：事实上，如果复习一下前面的章节，你就会发现我们十分明确地将这种技术称为“增量编码”而不是“增量压缩”，从技术角度来说，后者其实不正确，因为这里不存在压缩。

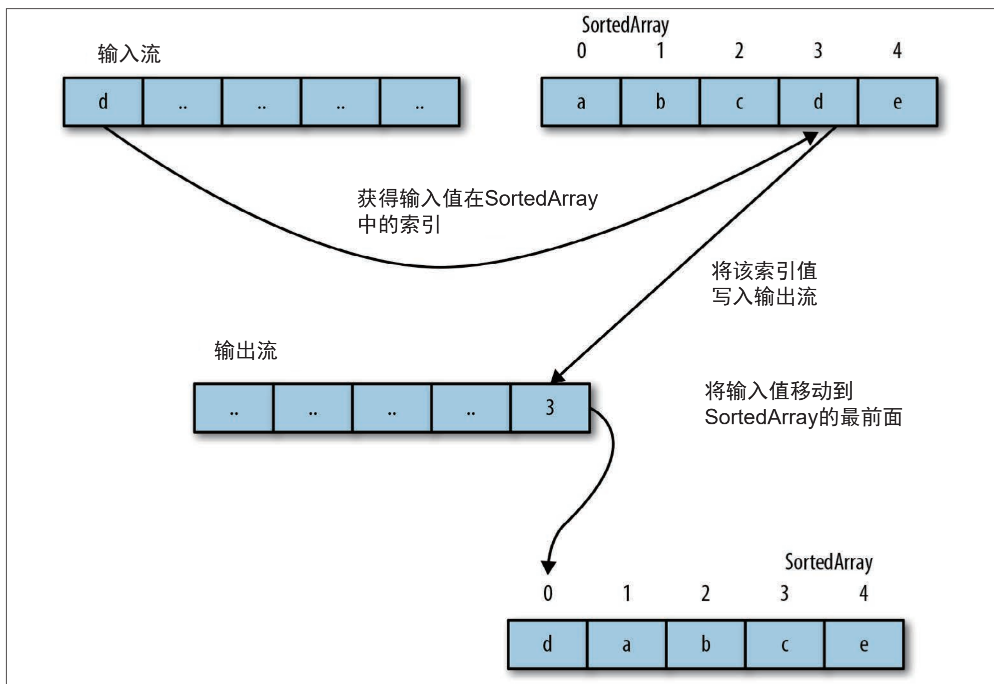


图 8-4：详细说明 MTF 是如何工作的原理图。当某个值在输入流中出现多次时，它就会移动到 SortedArray 的最前面，从而使输出的索引值相对较小

我们来看一个例子。为了简单起见，假定数据中的符号都是小写的 ASCII 字符，输入字符串为“banana”，初始状态下 SortedArray 中的字符是按字母顺序排列的。

(1) 读取字母“b”。

(2)“b”在 SortedArray 中的索引为 1[12]，因此将 1 写入输出流。

(3) 将“b”移动到 SortedArray 的最前面。

(4) 读取字母“a”，由于其现在的索引为 1，因此输出 1，同时将“a”重新移到最前面。

(5) 对其余的字母，按照顺序重复上面的过程，具体如下表所示。

输入符号	输出结果	SortedArray
		abcdefghijklmnopqrstuvwxyz
b	1	bacdefghijklmnopqrstuvwxyz
a	1,1	abcdefghijklmnopqrstuvwxyz
n	1,1,14	nabcdefghijklmopqrstuvwxyz
a	1,1,14,1	anbcdefghijklmopqrstuvwxyz
n	1,1,14,1,1	nabcdefghijklmopqrstuvwxyz
a	1,1,14,1,1,1	anbcdefghijklmopqrstuvwxyz

注 12：这个脚注是专门为非编程人员写的，数组的索引从 0 开始，因此 SortedArray 第二个位置的索引为 1。

最终的输出结果为 1,1,14,1,1,1，其信息熵为 0.65[13]，与源数据的信息熵 1.46 相比小很多。

解码器按照相反的步骤操作就能恢复源数据流。恢复时，输入流为 [1,1,14,1,1,1], 初始状态的 SortedArray 为 [a,b,⋯,z]，解码器每从输入流中读取一个符号，就将 SortedArray 中该索引对应的字符输出，再将该字符移到 SortedArray 的最前面。

8.3.1 消除捣乱符号的影响

MTF 存在的一个问题是，有一些捣乱的符号会打乱前面存在的符号流。这个问题比较严重，因为会极大地破坏编码，而且在真实数据中普遍存在。

一种解决方法是，不是一读到某个符号就将它移到最前面，而是采取一些探索式方法慢慢地将它移到最前面。通过下面这些例子，我们来看看探索式方法在实际中是如何很好地工作的。

1. 向前移动 *k* 个位置

如果采用这种方法，那么与当前符号匹配的元素就不再是直接移到 SortedArray 的最前面，而是将它前移 k 个位置。

这样就需要找出最佳的 k 值，有下面两种确定 k 值的简单方法：

- 令 $k = n$（n 为符号的个数），原始的 MTF 就是这种选择；
- 令 $k = 1$，即某个符号被读取一次，它的位置就前移一位。

将 k 取为 1 时，可能会对降低那些具有较好局部相关性的输入流性能，但是能更好地处理其他类型的输入流。而且，$k = 1$ 时算法实现起来也很简单，因为在更新 SortedArray 时只需要将当前读取的符号与前一个符号交换位置。这样设定之后，也能更好地处理那些捣乱符号，因为这些符号现在需要慢慢地移动到最前面，而不是一下子就移动到最前面。

2. 出现 *c* 次再移动

采用这种方法时，SortedArray 中的元素只有在输入流中出现过 c 次之后（并非必须连续出现），才会移动到最前面。SortedArray 中的每个元素都有一个计数器，记录该元素出现的次数。这样我们就可以为符号移动到最前面设定一个出现次数的阈值。当应用到文本上时，我们就可以通过最终生成的 SortedArray，来反映所编码的语言各个字母的使用频率。

8.3.2 压缩MTF

MTF 生成的输出流的熵通常要比源数据流小，这让它的输出成为传递给统计压缩器进行进一步压缩的首选对象。由于 MTF 生成的输出流中有很多的 0 和 1，因此简单的统计编码算法就可以工作得很好。

注 13：这样的结果可能真的会让香农头疼，这也是到目前为止本书中出现的最小的熵。

MTF 的独特之处在于，符号在短时间内重复出现时，MTF 会重新分配一个较小的值。而 RLE 会将最短的编码分配给那些连续重复出现的符号。实际上，MTF 是最简单的动态统计转换形式之一。

8.4 BWT

有规则就有例外，BWT 就属于例外。从前面的介绍中可以看出，所有其他的压缩算法通常可以归为两类：统计压缩（即 VLC）和字典压缩（如 LZ78），这两类算法从不同的角度利用了给定数据流中存在的统计冗余信息。

BWT 的工作原理并非如此。相反，它通过打乱数据流次序来让重复的子串聚集在一起。这一操作本身不能压缩数据，却可以为后续的压缩系统提供转换好的数据流，方便压缩。

BWT 是如何产生的

就压缩算法而言，BWT 的起源故事相当有趣。

1978 年，David Wheeler 在参观贝尔实验室时偶然发现了这一变换的最初版本，但当时，他没有考虑太多。Wheeler 最初的设想是将它当作基准，以与其他算法进行比较，因为他认为它运行得实在是太慢了。此外，Wheeler 所在的剑桥大学计算机实验室没有严格的出版要求，因此这一算法在此后 10 年的大部分时间里都没有被发现和报道。

Mike Burrows 在剑桥大学计算机实验室读研时，Wheeler 将这一算法教给了他，Burrows 只是将这一算法当成了写论文时学到的又一疯狂算法。当 Burrows 询问 Wheeler 他是如何创建这一算法时，Wheeler 却真的记不起来了。当时有一些基于上下文分组值的相关研究，但这一算法是如何实现可逆转换的，从来没有被揭示过。

直到几年后，Burrows 才认识到这一转换所做事情的重要性，同时意识到如果他不将算法写出来，那么这一算法可能永远不会被发表。因此，1994 年，Burrows 开始与 Wheeler 合作，利用当时的新技术和最新的计算机硬件开发了性能高效的实现。

本书作者柯尔特有幸在拍摄 *Compressor Head* 这一关于压缩算法的 YouTube 视频系列的第四集时见到了 Burrows 博士本人，Burrows 向他讲述了原始算法发表的经过。

“这其中有一个有趣的故事。我们先将论文投给了 *Data Compression Conference*，但遭到了拒稿，也没有给出拒稿的原因。当我去询问时，他们回复说没有解释拒稿原因的习惯。因此，我们将它作为技术报告发表了。有人看到了这份报告并在 *Dr. Dobbs Journal* 上发表了一篇关于它的文章后，这一算法就变得更出名了。第二年，*Data Compression Conference* 的人要求我再次提交这篇论文以便发表。我回复说：‘我不想发表，也不会解释为什么，因为我也没有解释这类事情的习惯。’”

你能想象吗？如果不是 Burrows 的努力让这一算法得以发表，可能世人永远也不会知道它。这再次证明了压缩世界是疯狂的，有些奇怪，还存有怨念。

8.4.1 顺序很重要

熵作为度量单位，它的一个问题是没有考虑符号之间的顺序。不管如何打乱符号集 [1234567890] 中符号的顺序，它的熵始终是 4。

但是我们知道，事实上符号之间的顺序很重要。例如，LZ 系列字典压缩算法就非常重视符号之间的顺序，本节介绍的其他上下文转换算法同样如此。

因此，如果顺序的确很重要，那么我们就有理由认为：通过转换数据流中符号之间的顺序，可以让数据流更容易压缩。

最简单的重新排序就是直接将数据按顺序排列，例如，将 [9,2,1,3,4,8,0,6,7,5] 排序为 [0,1,2,3,4,5,6,7,8,9] 后，我们就可以将其增量编码为 [0,1,1,1,1,1,1,1,1,1]，它的熵值要比源数据小很多。

不过遗憾的是，纯粹的排序是单方向的。也就是说，在对数据排序后，如果没有更多额外的信息指明它是如何变化的，我们无法让数据重新回到未排序的状态。

因此，我们不能单纯地对数据进行排序，但可以近似地这样做。

BWT 会打乱数据流中符号的顺序，并试图让相同的符号簇彼此靠近，这一行为通常称为**字典序排列**（lexicographical permutation）。或者更确切地说，通过 BWT，我们可以找出原始数据集的一种排列，根据其顺序，该排列可能更容易压缩。

其中最值得关注的一点是：通过 BWT，在编码与解码时无须增加太多的额外信息。下面我们来一探究竟。

8.4.2 BWT的工作原理

要开始 BWT 的“变换”工作，需要先创建一张表，其中包含输入流的所有移位排列。

举个例子，假定输入流是 BANANA，那么需要将它写在表的第一行。然后，在接下来的每一行，我们都会对该字符串进行一次循环右移一位操作。也就是说，将除了最右边的字符之外的所有字符向右移一位，然后将最右边的字符放在最前面。继续进行这样的移位操作，直到对所有的字符都操作了一遍，如下所示。

```
BANANA
ABANAN
NABANA
ANABAN
NANABA
ANANAB
BANANA
```

接下来，BWT 会对表中的每一行按字典顺序排序。看到所有的字符串又按照顺序排列，是不是感觉很好？

```
ABANAN
ANABAN
ANANAB
BANANA
NABANA
NANABA
```

现在，需要你注意每个字符串的最后一个字符（用斜体做了强调）。从上到下，这些字符组成了字符串 NNBAAA，有意思的是，这恰好是 BANANA 的一种排列，而且与 BANANA 相比更好地将相同的字符聚集在了一起。

事实上，这正是我们在找的排列。从上面的过程中可以看出，按照顺序循环生成排列，然后排序，这样由最后一列字符组成的排列，与源字符串相比，能更好地将相同的字符聚集起来。

因此，NNBAAA 就是字符串 BANANA 经 BWT 后输出的结果。

不过在你离开之前，还有一个重要的数据需要掌握。观察排序后的表格，你会发现输入字符串的索引为 3。

```
0  ABANAN
1  ANABAN
2  ANANAB
3  BANANA
4  NABANA
5  NANABA
```

在 BWT 的解码阶段，我们需要该索引值，因为它将使我们从更易压缩的排列回到源字符串上。

8.4.3　BWT的逆操作

BWT 最引人注目的特点并不在于它能生成更易压缩的输出（普通排序也能做到这一点），而在于只需要极小的数据开销，它所进行的变换操作就是**可逆的**（reversible）。

下面就来证实它的正确性。因此，我们的目标是对 BWT 解码，而所给的条件是字符串 NNBAAA 和行索引 3。

首先需要做的是重新生成排列表格。要做到这一点，就需要迭代利用排序和字符添加操作。

第一步将输出字符串写入下表中，它表示的是每一行的最后一个字符。

输出字符串/最后一列
[N]
[N]
[B]
[A]
[A]
[A]

说来也奇怪，如果对这一列排序，其结果与原来的排序后表格的第一列相同，如下表所示。[14]

输出字符串	排序后
[N]	[A]
[N]	[A]
[B]	[A]
[A]	[B]
[A]	[N]
[A]	[N]

接下来将这两列合并，这样每行就都有两个字符了：

```
[NA]
[NA]
[BA]
[AB]
[AN]
[AN]
```

按字典顺序对每行排序，结果如下：

```
[AB]
[AN]
[AN]
[BA]
[NA]
[NA]
```

接着，将原始的输出字符串（NNBAAA）按顺序添加每行字符串的最前面（每行添加 1 个字符），所得结果如下所示：

```
[NAB]
[NAN]
[BAN]
[ABA]
[ANA]
[ANA]
```

注 14：这不仅仅是由于 BANANA 这个单词的构造比较奇怪。这是我们观察到的 BWT 具有的性质，但至于为什么会有这个性质，连 BWT 的提出者也不清楚。

然后，再次对每行字符串按字典顺序按列排序，继续将原始字符串按顺序添加到每行字符串的最前面并按列排序，直到整个输出矩阵的宽度等于输出字符串的长度，具体过程如下表所示。

3	4	5	6	7	8	9
[ABA]	[NABA]	[ABAN]	[NABAN]	[ABANA]	[NABANA]	[ABANAN]
[ANA]	[NANA]	[ANAB]	[NANAB]	[ANABA]	[NANABA]	[ANABAN]
[ANA]	[BANA]	[ANAN]	[BANAN]	[ANANA]	[BANANA]	[ANANAB]
[BAN]	[ABAN]	[BANA]	[ABANA]	[BANAN]	[ABANAN]	[BANANA]
[NAB]	[ANAB]	[NABA]	[ANABA]	[NABAN]	[ANABAN]	[NABANA]
[NAN]	[ANAN]	[NANA]	[ANANA]	[NANAB]	[ANANAB]	[NANABA]

观察最后输出的矩阵，你很快就能发现如下两个神奇的性质。

- 最后的矩阵与在编码器中生成的排序后的置换矩阵完全相同。这意味着，即使只给出排序后矩阵的最后一列，NNBAAA，我们也能利用它来恢复生成其整个排序后的矩阵。
- 还记得在编码阶段输出的索引 3 吗？由于这个矩阵与编码器中排序后的矩形完全相同，因此只需要从矩阵中取出索引号为 3 的行（注意索引是从 0 开始的，因此索引 3 对应的是第四行），就能恢复源输入字符串 BANANA。

8.4.4 具体的实现

这里我们给你提个醒！

虽然拥有如此多的优点，但遗憾的是，不能在 50 GB 大小的文件上进行 BWT。这种置换变换的工作方式就决定了，我们每一行都需要存储 50 GB 大小的符号（每一列也是 50 GB 大小），并且按行依次左移一位符号。整个符号矩阵需要的空间太大了。

因此，我们通常将 BWT 称为块排序变换，具体实现时，它会将整个文件分为许多 1 MB 大小的数据块，然后在每个数据块上分别应用该算法。这样一来，大多数现代设备就能满足该算法对内存的要求，同时该算法也能获得较好的性能。

BWT 与 DNA

BWT 一直被当作一种边缘压缩算法。虽然最初在处理文本数据时，显示出非常好的效果，但是从性能角度来看，它永远无法与 GZIP 这样的算法去竞争。因此，BWT（或者说 BZIP2，主流的 BWT 编码器）从来没有在压缩世界掀起什么风浪。

情况一直如此，直到人类开始对脱氧核糖核酸（deoxyribonucleic acid，DNA）测序。

人类的 DNA 分子结构虽然复杂，但是其组成非常简单，只包含 4 种基本的核苷酸碱基，分别标记为 A、C、G 和 T。一个给定的基因组基本上可以看成是一个长字符串，包含这 4 个符号按不同的顺序排列。有多少这样的字符呢？据计算，人类基因组大约

包含 31.647 亿个 DNA 碱基对。

结果表明，BWT 这一块排序算法对 DNA 来说是一种理想的变换，可以使其更容易压缩、查询和检索。（事实上，有大量的论文证明了这一点。）当根据参照对新基因组进行读取校准时，大小的减小和可用性的提高对快速读取来说十分重要。

这又从另一个侧面说明，在数据压缩领域不存在无所不能的银弹。每个信息流都有其自身的变化特征，因此它对不同变换和编码器的响应也不同。虽然 BWT 没有能将网络世界从 GZIP 那里抢过来，但是这并不妨碍在未来的几十年里它将在生物信息学领域独领风骚。

8.4.5 压缩BWT后的数据

显然，BWT 本身不压缩数据，它只是转换数据。为了让 BWT 真正起作用，还需要应用其他的转换来生成熵值更小的数据流，然后再对其压缩。

最常见的算法是将 BWT 的输出作为 MTF 的输入，经过处理后接着用统计编码算法处理。这基本上就是 BZIP2 的内部工作原理。

1. 为什么不用 RLE

为什么使用 MTF 而不是 RLE 呢？这里需要记住的是 RLE 对干扰符号十分敏感，而 BWT 不能生成足够长的连续行程以确保 RLE 始终处于理想状态。相反，MTF 则对干扰符号这类问题不敏感。

2. 为什么不使用 LZ 算法

又为什么不使用 LZ 算法呢？下面来看一个简单的例子。需要记住的是，当字符串中出现较长的重复子串时，LZ 算法才工作得最好。

LZ 算法在字符串 TOBEORNOTTOBEORTOBEORNOT 上工作得很好，因为 TOBEORNOT 是其中最长的重复子串，但是，LZ 算法在其他类似的字符串上工作得并不好。

假定经过变化后得到的结果为“OBTTTTTTOOEER”。

(1) LZ 算法看到第一个 T 时会将其编码为字面值。
(2) 第二个 T 会被编码为前一个字符的匹配，其长度为 1。
(3) 遇到第三个 T 时会向前看一个符号，并将 TT 这组符号当成前两个字符的匹配，其长度为 2。结果是，6 个字符“T”会被编码为字面值 T 以及 < –1,1 > 、< –2,2 > 、< –2,2 > 这 3 个标记。

之后，如果再遇到一串很长的“T”值，我们可以期望出现一段较长的匹配。然而不幸的是，由于 BWT 使这些符号出现得这么分散，因此匹配的距离值可能会比较大，而这会影响我们对数据流的编码。

第 9 章

数据建模

任何玩过“传话筒”（telephone）游戏的人都知道上下文语境对大脑的重要性。在大多数情况下，单词“cup”（杯子）和“cop”（警察）本身出现的可能性大致相同。然而，如果是在一个喧闹的聚会上，你听到一个单词并认为它不是“cup”就是“cop”，那么你的大脑就会根据前面的上下文来确定它会是哪个单词。例如，如果你新认识的朋友说“Wash the”（洗），那么下一个单词就很有可能是“cup”；如果他说“Run from the”（快跑），那么下一个单词就可能是“cop”。[1]

这就是多上下文编码算法背后的基本概念。它会考虑最后观察到的几个符号以确定当前符号的理想编码位数。

也许，一个更具体的例子是英语中的字母组合如何影响后面字母的出现概率。

例如，在“典型”的英语文本中，字母“h”平均的出现概率大约是 5%。然而，如果当前字母是“t”，那么下一个字母是“h”的概率就会高很多，其出现概率大约为 30%，这是因为“th”这样的字母组合在英语中很常见。类似地，字母“u”平均的出现概率大约是 2%。而如果当前字母是“q”，那么下一个字母是“u”的可能性则会超过 99%。在这个例子中，通过当前字母是“q”，我们就能预测到下一个字母会是“u”，因此可以分配给它更少的二进制位数。这种基于统计观察的相邻关系，通常也被称作“预测”编码器，你会在大多数

注 1：不过，如果是在互联网上，也可能会有用到“洗警察”或者“快跑，杯子”这样句子的地方。（谁知道呢？）

正式的压缩文献中看到这个“恰当”的术语。[2]

这类编码器也可以认为是统计压缩器的“加强”（on-steroids）版。它将自适应模型（见第 3 章）和多种符号码字对应表（见第 2 章）结合起来，根据前面观察到的符号，为当前符号生成尽可能短的码字。

不过这并不是什么新的概念。早在 18 世纪[3] 它就被首次提出，迄今还是最强大的统计计算工具之一。

9.1 马尔可夫链

马尔可夫链（Markov chain）这个概念很有意思，我们先来看一下其比较迷惑人的技术性定义：

马尔可夫链是一种离散的随机过程，其未来的状态只取决于现在，而与过去的历史无关。

有了这样的定义，假定有一个学生已经完成了中学三年级的课程，而我们想知道他在中学四年级的数学课上得 A 的概率。一般来说，通常我们会认为这样的预测会取决于他在中学一年级、二年级和三年级的数学课上所取得的成绩。然而，如果只有三年级（即当前）的成绩会对结果产生影响，而前两年的成绩完全可以忽略不计，那么就可以认为这是一个马尔可夫过程。

我们再来看一个更详细的示例。

假定我们刚刚度过了长达 104 天的暑假，现在事后回顾[4]，我们想分析一下这个暑假是怎么过的，还想根据当天是周几来分析。我们发现这个暑假中每周一进行洞穴探险和地掷球中任何一种活动的可能性都是 50%，如下表所示。

星期	活动	概率
周一	洞穴探险	50%
周一	地掷球	50%

注 2：注意，在常见的压缩文件中，这些被称为“预测”编码器。不过作为本书的作者，我们不喜欢这个术语，因为“预测”就意味着“可能会出错”。这似乎违背了人们对压缩的直觉认知，如果在编码或者解码时出现预测下一个符号错误的情况，那么最终得到的会是一个损坏的压缩系统。相反，我们更喜欢称这些算法为多上下文算法，因为它将多个符号和统计表 / 模型结合在一起，以确定编码下一个符号所需的最少二进制位数。

注 3：决策支持系统的历史可以追溯到帕斯卡，参见维基百科 Pascal 词条。

注 4：对那些阅读数据压缩方面图书的人来说，暑假评估是一件再平常不过的事情。

可以用如下的术语来描述这一发现：如果今天是周一，那么进行洞穴探险和地掷球中任何一种活动的概率都是 50%。

我们发现周二的活动种类更多，具体情况如下表所示。

星期	活动	概率
周二	在泳池边躺着	10%
周二	跳袜子舞[a]	20%
周二	修剪草坪	30%
周二	洞穴探险	40%

a 这是一件大事，或者更确切地说，在过去这是一件大事。

也可以用类似的语句描述周二，即“如果今天是周二，那么跳袜子舞的概率为 20%”。

实际上，这就是我们所说的“二阶上下文”（second-order context）。我们有两组数据并利用它们来定义某个活动的发生概率。星期在这里是作为“一阶”数据（“first order”或“context-1”data），相应的活动则是“二阶”数据（“second order”或“context-2”data），其结果是百分比形式的概率。

下面来看一下三阶的上下文，如下表所示：

星期	第一个活动	第二个活动	第二个活动的发生概率
周一	地掷球	修脚	5%
周一	地掷球	吃冰沙	95%
周一	洞穴探险	旧金山 HTML5 聚会	50%
周一	洞穴探险	吃比萨饼	25%
周一	洞穴探险	缝纫课程	25%

这个例子稍微有些复杂，我们来看表格的第 4 行，意思是说“如果今天是周一，**并且**我们刚刚进行了洞穴探险，那么接着去吃比萨饼的概率为 25%”。

每个上下文都在某种程度上描述了状态之间的转移。

也可以将它可视化为一棵树（见图 9-1），这里每个节点代表一个活动，每次转移则有相应的概率。

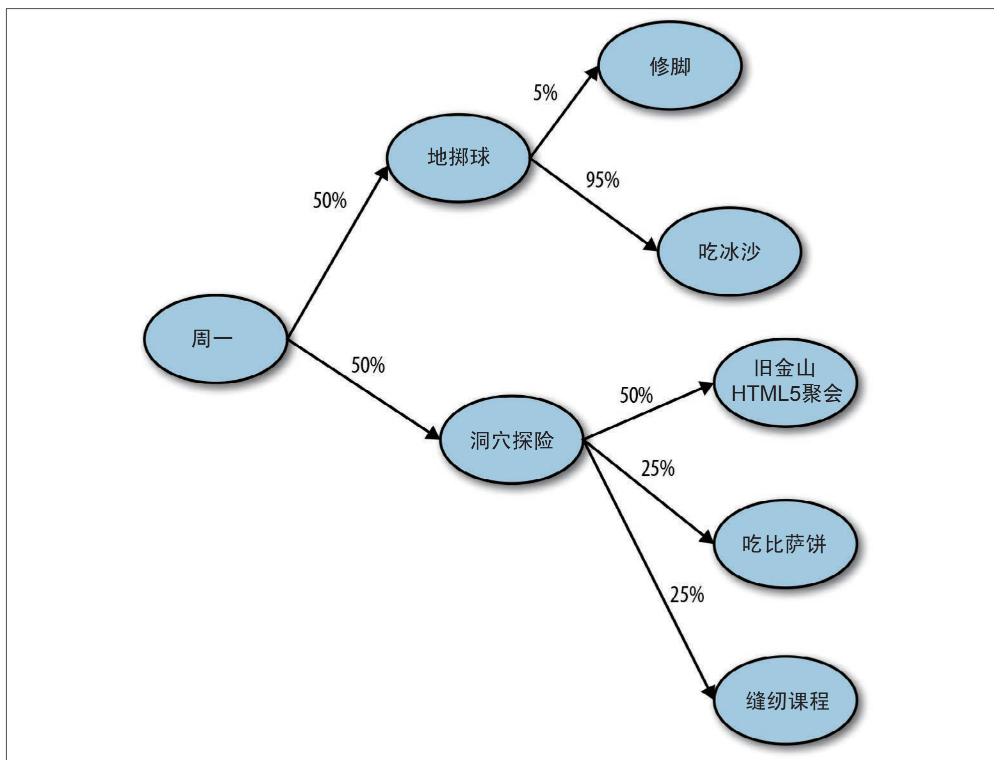


图 9-1：马尔可夫链工作方式的树形图

走近马尔可夫

1913 年，安德烈·安德耶维齐·马尔可夫（Andrey Andreyevich Markov）通过将数学与诗歌结合起来，创立了概率论的一个新分支。

在钻研亚历山大·普希金（Alexander Pushkin）的长篇诗体小说《叶甫盖尼·奥涅金》的过程中，马尔可夫花了大量时间详细分析元音和辅音出现的规律。1913 年 1 月 23 日，他发表了自己的研究成果，建立了一个统计模型详细说明，给定一个字母，接下来会出现哪个字母有着有限且可重复的概率。

大多数人认为，马尔可夫很勇敢。对于对手来说，马尔可夫非常好斗，他经常参与到针对当局的公共抗议和斗争中，而且因为需要很多天才能从打斗中恢复而闻名。当研究成果发表时，他已年过半百并且退休好几年了。历史上最强大的统计模型之一的提出者是一个无由的反叛者，这得有多合适啊！

马尔可夫提出的事件选择概率（probabilistic event selection）这一概念，与当时概率界的主流观点格格不入，当时的概率模型大都与抛掷硬币或掷骰子有关。马尔可夫链则帮助我们提出了关联概率的问题，例如“如果今天多云，那么明后两天会下雨的概率是多少？”在 1913 年，用数学解决这个问题的准确性与投掷鸡骨头瞎猜差不多。

最近几年，巨型马尔可夫链得到了应用。例如，由谷歌的创始人 Larry Page 和 Sergey Brin 设计的 PageRank 算法就是以马尔可夫链为基础，其中的状态就是互联网上总数大约 400 亿的网页，而转移则是网页之间的链接。这个算法就是为了计算如果用户随机浏览，那么他看到每个网页的概率是多少。

亚马逊通过马尔可夫链来决定向用户推荐什么样的产品。例如，如果其他人在浏览 A 商品后购买了 B 商品，那么当你浏览 A 商品的时候，亚马逊向你推荐 B 商品的可能性就会很大。

马尔可夫链在游戏推广方面的应用前景也很广阔，很多游戏厂商这样认为：“如果你以前喜欢动作类的游戏，而现在又喜欢与小狗有关的游戏，那么就可以得出：如果出现一款新的小狗对抗外星侵略者的动作游戏，你一定会很喜欢。”[5]

总的来说，这些算法在搜索与寻找、预测天气以及匹配用户喜好方面的能力是无穷的，甚至有可能通过这些算法来预测下一次军事冲突的情况。

9.1.1 马尔可夫链与压缩

马尔可夫链这一概念能很好地融入现有的模型，因为可以认为统计编码算法就是一阶马尔可夫链。有了数据流中各符号出现的概率表，我们就能为其分配相应的码字。

通过为每个在前面出现的符号增加一张符号码字对应表，我们就可以创建二阶马尔可夫链，如图 9-2 所示。下面来看看它的工作原理。

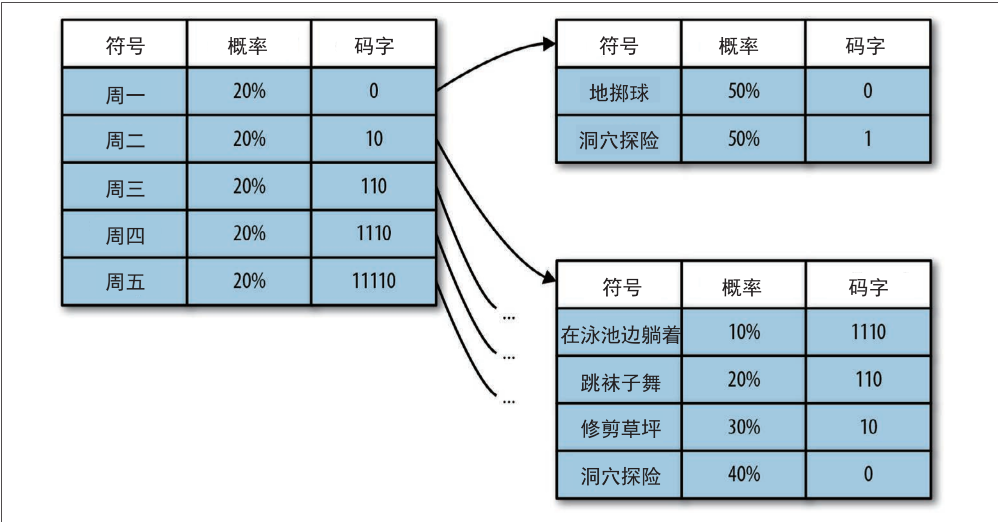


图 9-2：用符号表树表示的二阶马尔可夫链，数据源于前面所举的暑假活动的示例

注 5：事实上，这是本书中唯一一段两位作者不能达成一致的内容，因为其中一位作者认为，很明显，拯救这个世界的应该是可爱的小猫。

根据图 9-2，“周一，洞穴探险；周二，在泳池边躺着”编码后的结果为 0 1 10 1110，总共 8 个二进制位。相比之下，如果想将 10 个状态都遍历一次，那么编码同样的数据最终需要的二进制位数肯定会超过 12。

从技术角度来看，为压缩而建立的马尔可夫链遵循的许多规则与前面介绍的自适应统计编码（参见第 2 章）相同，比如读取符号之后再动态更新频率表，等等。

1. 编码

我们以为字符串“TOTOTO”创建马尔可夫链为例来说明整个编码过程。

(1) 首先创建一张仅包含字符 < literal > 的一阶表，其出现概率为 100%，具体如下表和下图所示。

一阶字符（符号的整体出现概率）	出现频率	码字
< literal >	100%	0

输入：**TOTOTO**

↑

(2) 接着读取第一个字符，这是一个新符号“T”。

(3) 更新一阶表使其包含字符“T”，并相应地调整字符的出现概率，调整后的表如下所示。

一阶字符		
字符	出现频率	码字
T	50%	1
< literal >	50%	0

(4) 输出 < literal > 对应的码字和字符“T”，如下图所示。

输出流：0 T

输入：T**OTOTO**

↑

(5) 从输入流中读取下一个字符，其值为“O”，它同样是一个新符号。

(6) 更新一阶表使其包含字符“O”，并再次调整字符的出现概率。

(7) 接下来调整符号的码字，以包含所有的字符并使其满足前缀性质，调整后的表如下所示。

一阶字符		
字符	出现频率	码字
T	33%	0001
O	33%	001
< literal >	33%	01

(8) 输出 < literal > 对应的码字和字符“O”，现在的输出流为 0 T 01 O，如下图所示。

输出流：0 T **01 O**
输入：TO**TOTO**



(9) 继续读取下一个字符，其值为“T”，它已在一阶表中。

(10) 更新一阶表中各字符的出现概率。

(11) 交换一阶表中字符对应的码字，使出现概率最大的字符其码字最短，交换后的结果如下表所示。

一阶字符		
字符	出现频率	码字
T	50%	01
O	25%	001
< literal >	25%	0001

(12) 将字符“T”对应的码字输出，其值为 01，现在的**输出流**为 0 T 01 0 01，如下图所示。

输出流：0 T 01 O **01**
输入：TOT**OTO**



(13) 继续读取下一个字符，其值为“O”。

(14) 更新一阶表中各字符的出现概率，并保持其对应的码字不变。

一阶字符		
字符	出现频率	码字
T	40%	01
O	40%	001
< literal >	20%	0001

(15) 将字符“O”对应的码字 001 添加到输出流，这样当前的**输出流**为 0 T 01 O 01 **001**。

(16) 现在，通过建立第二个符号码字对应表来创建马尔可夫链中的二阶链接，它表示的是“T”后的符号，如下表和下图所示。

二阶字符（“T”后的符号出现概率）		
字符	出现频率	码字
O	50%	0
< literal >	50%	1

输入：TOTO**TO**



(17) 继续从输入流中读取下一个符号，还是“T”。

(18) 更新一阶表中字符的出现概率并保持其对应的码字不变，如下表所示，“T 值之后”的二阶表则保持不变。

一阶字符		
字符	出现频率	码字
T	50%	01
O	33%	001
<literal>	17%	0001

(19) 将“T”对应的码字 01 输出，当前的**输出流**为 0 T 01 O 01 001 **01**。

(20) 现在，为“O”后的符号建立一张二阶表，如下表和下图所示。

二阶字符（“O”后的符号出现概率）		
字符	出现频率	码字
T	100%	0

输入：TOTOT**O**

(21) 读取最后一个符号“O”。

(22) 这里，可以利用“T 后”的二阶表，并输出 0 作为最后一个符号，因此**最终的输出流**为 **0 T 01 O 01 001 01 0**。

瞧，我们想要的压缩！

2. 解码

下面对最终的输出流进行解码，以验证上面的步骤确实可行。

(1) 读取 0，我们知道这是一个字面值符号，因此建立一张一阶上下文表（概率为 100%，以下都将省略百分号，只写数字）。

    0 T 01 O 01 001 01 0

(2) 读取字符“T”，它同样是个字面值，将它加入一阶表并更新字符的出现概率（两个字符出现概率为 50/50），并**将 T 输出**。

(3) 读取“01”，根据一阶表，它还是个字面值。

(4) 读取“O”，更新一阶表（3 个字符出现概率为 33/33/33），并**将“O”输出**。

(5) 读取“01”，根据一阶表，其对应的字符为“T”，**将“T”输出**并再次更新一阶表中字符的出现概率（现在为 3 个字符的出现概率为 50/25/25）。

(6) 读取“001”，根据一阶表，其对应的字符为“O”，**将“O”输出**并更新一阶表中字符的出现概率（现在为 3 个字符的出现概率为 40/40/20）。

(7) 对“T”之后的符号建立二阶表。

(8) 读取“01”，其对应的字符为“T”，**将“T”输出**并更新一阶表中字符的出现概率（现在为 3 个字符的出现概率为 33/33/33）。
(9) 对“O”后的符号建立二阶表。
(10) 读取“0”，前面的字符是“T”，在“T 后”的二阶表中寻找 0，得到“O”，**将“O”输出**。

这样就将输入流解码为源字符串。

当面对更多的符号、更长的字符串时，同样可以这样做，而且正如你看到的那样，情况很快就会变得很复杂。

3. 压缩

既然如此复杂，那么又如何在压缩方面有所收获呢？

当应用到压缩上，马尔可夫链可以让我们用更少的二进制位数对相邻的符号编码。

我们来看看二阶表，两个二阶表的码字中都包括 0 这一最短的可能码字。由于这里使用了前面的符号来区分上下文，因此同样的 VLC 可以使用两次，从而节省了空间。换一种说法就是，每个上下文都有自己的 VLC 空间，所以可以使用同样的 VLC。

如果遇到更多的符号和更长的输入流，我们就可以建立更多阶的上下文。以英语为例，T 是一个上下文，当它后面跟着 H 时，TH 就又是一个上下文，当其后跟着 E、I、U、O 和 A 时，又会形成新的上下文。

虽然字母 U 出现的概率只有 2.7%，这意味着如果不考虑上下文的话，分配给它的码字就会比较长，但是如果有了“出现在 Q 后”这个特定的上下文，分配给它的码字就会短很多。

这使得马尔可夫链格外强大。

9.1.2 实际的实现

值得指出的是，没有人真正地使用马尔可夫链来压缩数据，至少不会用前面说的方法来压缩。

下面考虑一种很坏却并非不可能的场景，假定我们遇到了一个八阶上下文的马尔可夫链。

这意味着对每个节点来说，它的深度都为 8 且拥有 256 个子节点。也就是说，需要 256^8 字节或者说 16 个**艾字节**[6] 来表示这张表。这个数实在是太大了，即使是按照现代计算标准来说也太大了[7]。正因为如此，人们创造了各种衍生算法，这些算法所需要的内存要比一般的马尔可夫链算法小，性能也更好一些。其中，最值得关注的是**部分匹配预测算法**和**上下文混合算法**，下面来介绍这两种算法。

注 6：即 17 179 869 184 GB。

注 7：写这句话的时候还是 2015 年，当时双核移动设备还是这个地球上的绝对主流。

9.2 部分匹配预测算法

要使马尔可夫链算法变得实用，就必须要解决内存消耗问题与计算性能问题，即使用最佳链来编码。John Cleary 与和 Ian Witten 于 1984 年 [8] 提出了马尔可夫链算法的一种具体实现，并称之为**部分匹配预测算法**（prediction by partial matching，PPM），该算法在内存消耗与计算性能方面表现都还不错。与马尔可夫链类似，PPM 算法同样通过前 N 个符号的上下文来决定第 $N+1$ 个符号最有效的编码方式。

一般来说，简单马尔可夫链会采用比较直接的实现方式，即读取当前符号并判断它是否是现有链条的延续，PPM 算法的实现恰好相反。给定输入流中的当前符号，PPM 算法会向前扫描 N 个符号，并根据前 N 个符号的上下文来决定当前符号的出现概率。如果在 N 个符号的上下文中，当前符号的出现概率为 0，PPM 算法就会将上下文符号的个数减少为 $N–1$。如果没有在任何上下文中发现匹配，就会做出固定的预测。下面举个例子。

(1) 假定我们在压缩某个输入流时遇见“HERE”这个单词好几次，因此“HERE”就变成了上下文之一。
(2) 在之后的某个地方，编码器又遇到了“THERE”并且当前正在压缩的是 R 字符。
(3) 在三阶（符号）上下文中，R 之前的符号是“THE”。
(4) 然而，编码器从来没有见过“THER”，只见过“THE ”（E 后有一个空格）。
(5) 因此，当前字符 R 出现的概率为“0”。（也就是说，在给定前 3 个符号的情况下，R 从来没有出现过。）
(6) 此时，PPM 算法会尝试二阶（符号）上下文，将“HER”匹配为一个链条。
(7) 这样一来就成功了，因为“HERE”在此之前已出现过多次，为“HE”创造了二阶上下文的语境。
(8) 因此，基于二阶上下文链“HE”，字符 R 的出现概率不再为 0。

接下来，我们来看看更正式的步骤描述。

(1) 编码器从输入流中读取下一个字符“S”。
(2) 编码器扫描最近读取的 N 个字符，也就是 N 阶的上下文。
(3) 根据最近读取的这些字符，编码器确定字符“S”出现在特定的上文后的概率 P。
(4) 如果这一概率为 0，编码器就会输出转义标记，这样解码器就能反映这一过程。[9]
(5) 然后，编码器继续从第二步的扫描操作开始，只不过扫描的字符个数由 N 变为 $N–1$，直到所得的概率 P 不为 0 或者没有字符可以继续扫描。

注 8：参见 J. Cleary 与 I. Witten 的论文 *Data Compression Using Adaptive Coding and Partial String Matching*，载于 *IEEE Transactions on Communications*，1984 年，第 21 卷，第 4 期，第 390~402 页。

注 9：是的，你理解得完全正确，这意味着如果一个符号之前没有出现过，那么 PPM 算法会在输出最后一个字面值符号之前将 N 个“转义码”输出。这一算法很多变体（包括 PPMA、PPMB、PPMC、PPMP 和 PPMX）之间的主要区别，就在于它们在处理转义码过程中的细微差别。

(6) 如果没有字符可以继续扫描，就为 *P* 赋上固定的概率值（例如基于字符出现频率的概率值）。

(7) 最终编码器通过统计编码的形式以概率 *P* 编码字符“S”。

9.2.1 单词查找树

要实现 PPM 算法，我们遇到的首要问题是如何创建一种数据结构，可以将从输入流中读取的每个字符的所有上下文（0~*N* 阶）存储起来，并且在需要时能快速定位到。在简单的情况下，可以通过一种被称为单词查找树（trie[10]）的特殊树结构来实现这样的需求，这种树的每个分支都表示一种上下文。

下面以字符串“ABAC”为例，来创建一棵 PPM 单词查找树，该字符串所允许的最大上下文[11]为二阶，如图 9-3 所示。

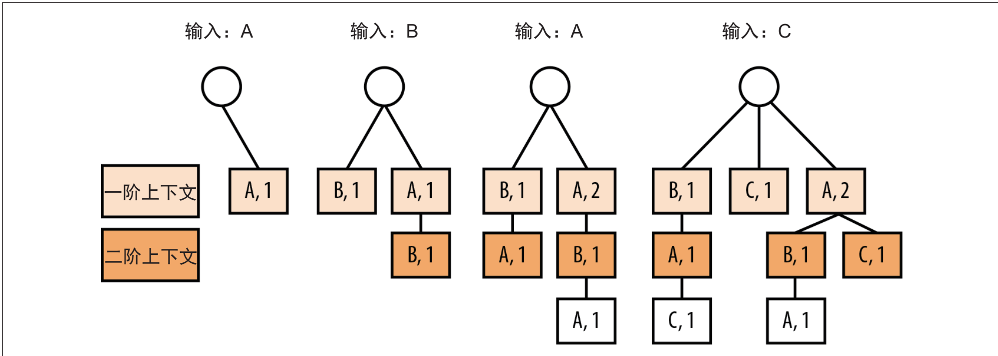


图 9-3：二阶上下文条件下创建的 PPM 单词查找树，树的每一层都表示一个上下文，根节点的子节点表示的是一阶上下文，字符右边的数字表示的是在此上下文中该字符出现的次数

(1) 读取第一个字符“A”，将节点 [A,1] 作为子节点添加到树的根节点上，表示的意思是字符“A”在一阶上下文中已经出现过一次。（根节点的子节点代表的是一阶上下文，根节点的孙节点代表的是二阶上下文，以此类推。）

(2) 读取第二个字符“B”，此时需要为单词查找树增加两个子孙节点。

a. 首先增加一阶上下文节点 [B,1]，当“B”是上下文链的首字母时，这样做是有用的。

b. 其次还需要在一阶上下文节点 [A,1] 下增加一个二阶上下文节点 [B,1]，它表示的是迄今为止从输入流中读取的字符链“AB”。

(3) 读取第三个字符“A”，这里有一些值得注意的地方。

a. 首先，需要将一阶上下文节点“A”的出现次数更新为 2（因为一阶节点 A 在之前已出现过）。

注 10：这个词就是这样拼写，请相信我们并找出它这样拼写的原因。

注 11：限制上下文的阶数是控制复杂性的一个方法。

b. 还需要将 [A,1] 作为每个 [B,1] 的子节点添加到树上，它们分别表示的是一阶上下文“BA”和二阶上下文“ABA”。

(4) 读取最后一个字符“C”，还是依照前面的过程增加子孙节点，不过有一些小的不同。

a. 首先，增加一阶上下文节点 [C,1]。

b. 下一步是在所有的二阶节点 [A,*] 下增加三阶节点 [C,1]。可以看到最左边从 [B,1] -> [A,1] 这条链上，[C,1] 顺利地加上去了，但是在右边增加新节点的话就会超过前面规定的上下文高度，因此这里没有增加新节点。

c. 最后一步，将 [C,1] 添加到一阶上下文节点 [A,2] 下作为它的子节点。

创建这样的查找树是有用的，给定当前的状态，我们很快就能将 *N* 减 *X* 阶上下文查找出来。例如，假定当前字符是“C”，那么它的二阶上下文是“BA”，一阶上下文是“A”。这与实际完全符合，因为它表示的就是编码“ABAC”时在“C”之前的一个和两个符号的滑动窗口。

在限定二阶上下文的情况下，可以用这棵查找树来表示输入字符串的所有子字符串（如图 9-4 所示）：B、BA、BAC、C、A、AB、AC 和 ABA。

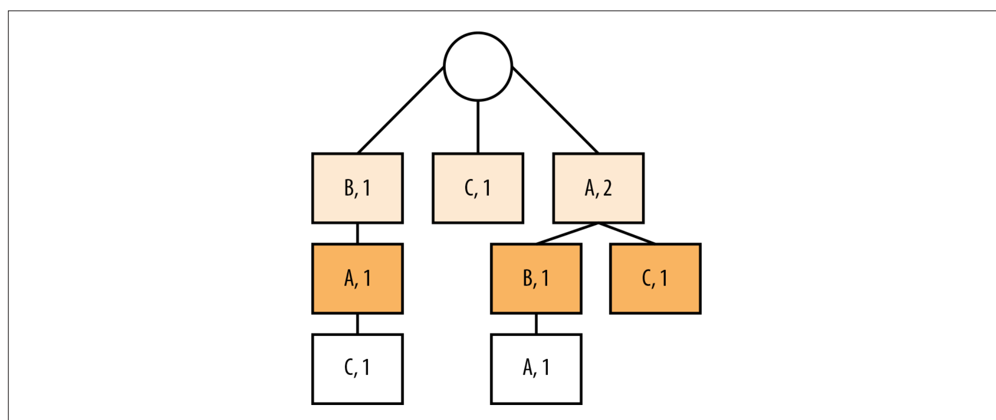


图 9-4：查找树表示的子字符串，B、BA、BAC、C、A、AB、AC 和 ABA

9.2.2 字符的压缩

除了能提供高效的存储以及快速提取子字符串外，单词查找树的每一层都会记录字符出现的次数。有了这些数据，统计编码算法就能构建出字符出现的概率表，并根据概率表为每个字符分配相应的码字，如图 9-5 所示。

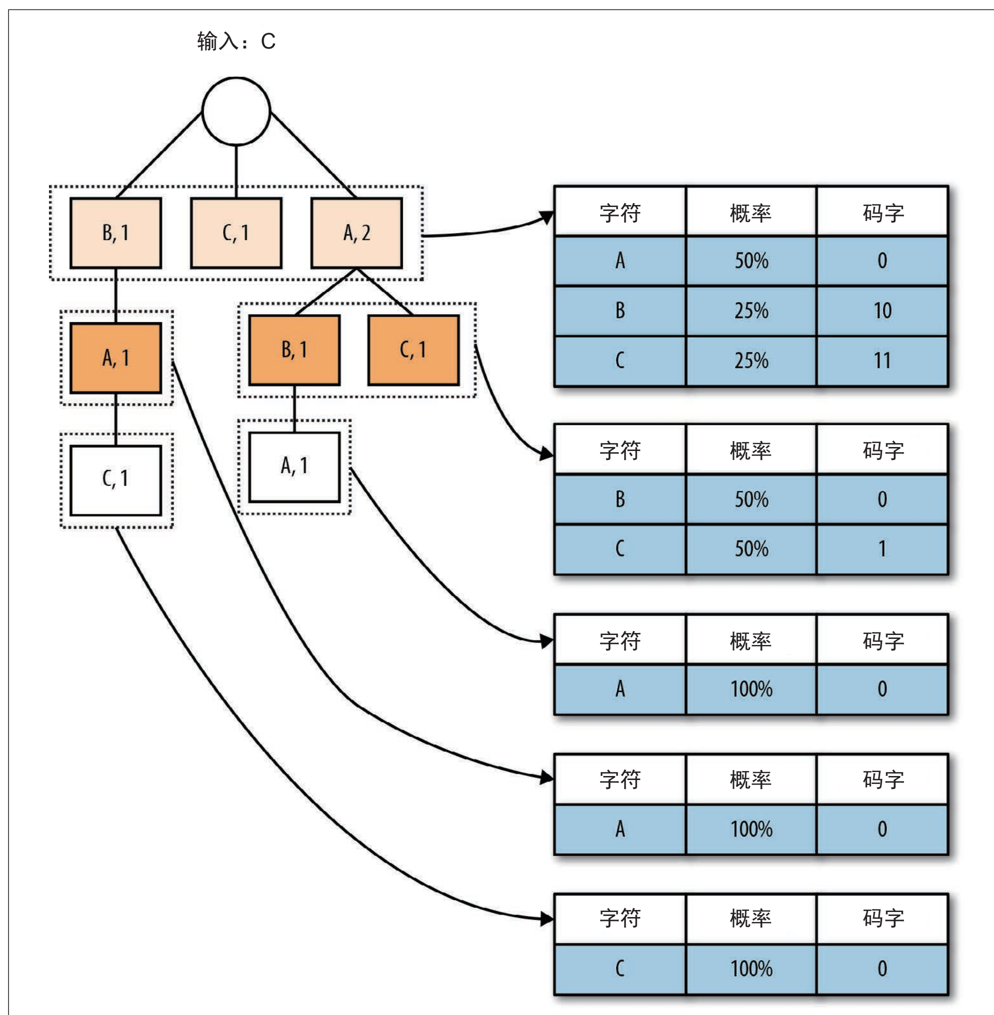


图 9-5：PPM 算法单词查找树以及可以用来编码的符号码字对应表示例

实际上，对树的每一层，只需要考虑这一层的兄弟节点并将出现次数归一化就能得到相应的概率。例如，一阶上下文中节点 [B,1]、[C,1] 和 [A,2] 中出现次数对应的概率分别为 25%、25% 和 50%。

此外，注意因为在二阶上下文中编码每个字符都只用了 1 个二进制位，所以每个字符都节省了 1 个二进制位。

只要将第 3 章介绍的数据结构改一改，你就能以简单的方式达到上述效果。

9.2.3 选择一个合理的*N*值

那么 *N* 值应该是多少呢？ PPM 算法会选择一个 *N* 值，然后再根据这样的上下文长度去寻找匹配。如果没有找到匹配，就会选择更短的上下文继续寻找。这样看来，似乎上下文越长（也就是 *N* 的取值越大），预测的效果越好。然而，大多数 PPM 算法的实现在综合考虑所需内存、处理速度以及压缩率后，将 *N* 的值设定为 5 或者 6。

也有一些 PPM 算法的变体，例如 PPM*，尝试着使 *N* 的取值变大，并且是变得非常大。这样做，不仅需要一种新的查找树结构，而且需要的计算资源也远比 PPM 多，但其结果则比 PPM 算法好，能多节省约 6% 的存储空间。

9.2.4 处理未知的符号

PPM 算法（模型）的大部分优化工作是在处理输入流中那些之前没有出现过的字符。显而易见的处理方法是通过创建“从没见过”的符号来触发转义序列（escape sequence），但是那些没有见过的符号应该赋什么样的概率值呢？这通常被称为**零频问题**（zero-frequency problem）。

有一种 PPM 算法的变体使用拉普拉斯估计（Laplace estimator），赋给所有“从没见过”的符号相同的伪计数值 1。还有一种被称为 PPMD 的变体是这样处理这一问题的，“从没见过”的符号每使用一次，伪计数值就加 1。（换句话说，PPMD 算法是这样估算新出现符号的概率的，即新符号的概率等于不同符号的个数与观察到的所有符号的出现次数之比。）

PPMZ 算法的处理更有意思。刚开始时它的处理方式与 PPM* 相同，都试图在 *N* 阶上下文下找出当前符号的匹配。当找不到这样的匹配时，它就会换上完全不同的算法**局部阶估计法**（Local-Order-Estimator），而使用的还是基本的 PPM 模型，只不过预测的算法完全不同。

9.3 上下文混合算法

对 PPM 算法的改进，让我们有了新的数据压缩算法，PAQ 系列算法。特别是在 PPMZ 算法中，对于符号如何去响应匹配，人们尝试了多种类型的上下文。

随着时间的推移，这一概念逐渐被称为上下文混合算法（context mixing），即为了找出给定符号的最佳编码，我们会使用两个或者更多的统计模型。例如，通过一个统计模型来预测在所有的经常性活动（比如营救小猫之类）中你去健身房的概率有多大，用另外一个统计模型来预测吃了太多意大利面后的 12 个小时内你去健身房的概率。在面对“现在你要去健身房的概率有多大”这样的问题时，每个模型会给出不同的概率。由于通常情况下你去健身房的概率为 20%，而且离你吃完意大利面已经有 6 个小时了，因此现在去健身房的概率很可能是 50%。上下文混合算法就是综合去利用这些概率。

关于数据压缩，上下文混合算法也带来了以下两个很有意思的问题。

- 应该使用什么样的模型？
- 应该怎样将这些模型结合起来？

9.3.1 模型的类型

正如前面讨论的那样，对数据压缩来说，相邻性是一个很重要的课题。LZ、RLE、增量编码以及 BWT 这些算法都是基于这样的假设：数据的相邻性与它的最佳编码方式有关。

在介绍马尔可夫链时，我们也很容易持有同样的观点。创建基于相邻性的上下文（诸如“A 位于 B 后”之类）很容易，但实际上，相邻性只是在符号之间建立上下文相关的一个方法。例如，你可以创建一个所有下标值为偶数的符号上下文，或者创建取值聚集在某个数值范围内的符号上下文。总的来说，相邻性和局部性都只是上下文的最简单形式，而绝不是唯一的形式。

有了这种认识后，去数据流中寻找一些与上下文以及相邻性完全没有关系的其他信号，来帮助我们找出编码当前符号的最佳方法，也就很容易从逻辑上理解了。我们所说的模型其实就是用来识别和描述符号之间的关系。通过对数据的建模，我们就能对数据中包含的各种属性了解得更多，因而也就越能描述好当前的符号。

实际上，模型可以有很多种，需要处理的数据类型不同，模型也会不同。

例如，图像数据更多地关注二维的局部性，也就是说一个像素的颜色通常与上下左右四个方向相邻像素的颜色有关，可以利用这一信息对图像进行压缩。然而，这一模型不适用于文本，在文本中通常找不出这样的规律，即一个字符与其上一行或下一行同样位置的字符无关。[12]

在编程中，当把高级的指令编译为字节码后，就会呈现出完全不同的模型。一个字节可以描述指令，后面跟着长度不等的字节描述函数的输入。由于代码往往有相同的模式，因此你完全可以按照下面的方式建模：当看到“跳转到此指令”命令后，你极有可能会看到附近出现“将变量压入调用栈”之类的命令。因此，在这种情况下，相邻的字节不重要，重要的是命令本身。

音乐是另外一种完全不同的数据，你可以建立模型来表示低音旋律或者吉他即兴重复，并考虑其中涉及的音阶或者变奏过渡的长度。

关键在于，你如果对数据足够了解并且能正确地提问，就会有成千上万种建模的方法。因此，在某些情况下问题的难度就增加了，因为现在讨论的不再是通用算法，而更多的是提

注 12：除非你正在压缩的文本是文字游戏或是一些奇怪的诗歌。

问："你是否对数据了解得足够多，可以正确地对其进行建模？"

作为上下文混合算法的先驱压缩器之一，PAQ 包含以下模型：

- *n* 元语法（*n*-grams），这里上下文是指在被预测符号之前的 *n* 个字符（与 PPM 算法相同）；
- 整词 *n* 元语法（whole-word *n*-grams），忽略大小写和非字母字符（对文本文件很有用）；
- "稀疏"上下文，例如，被预测符号之前的第二个和第四个字节（对某些二进制文件很有用）；
- "模拟"上下文，由前面的 8 位或者 16 位字节的高字节位组成（对多媒体文件很有用）；
- 二维上下文（对图像、表和电子表格很有用），行的长度由找出的重复字节模式的步长决定；
- 只针对特定文件类型的特殊模型，例如 x86 可执行文件，BMP、TIFF 或者 JPEG 格式的图片。

PAQ 不是小打小闹，它在大文本压缩基准测试（Large Text Compression Benchmark）中经常排名靠前，它的最新版本之一 ZPAQ 在压缩人类 DNA 信息的比赛中获得了第二名。

9.3.2 混合的类型

随便一问，给定两个值，你会怎样将它们综合起来呢？取它们的平均值？还是算两者之和？抑或根据喜好或者之前的输入赋以不同的权重？当你开始使用多个模型时，这类问题会变得更加复杂。如果面对的是一组 50 个输入，你又会怎样将这些模型结合起来以达到最好的压缩效果呢？

幸运的是，统计学已经基本上帮我们解决了这个问题。将不同模型的输出结合起来有以下两种方法。

一种是**线性混合**（linear mixing），它是将各个模型的预测值加权平均的过程，最终的值则取决于证据权重。

在前面举的例子中，我们想知道的是你现在去健身房的概率，而给定的值是你多久去一次健身房以及吃一次意大利面。这两个给定的值有不同的证据权重，其中的一个值由于有更多的试验数据或者说样本数据，因而更可信。例如，如果你是在人的一生这个时间层面考虑意大利面如何影响你去健身房，那么就会有更多的样本数据，因此相应的取值也就比你上周去健身房的次数更可信[13]。因此，对这样的值我们会在混合模型中给予更高的权重，因为它更可信。

相比而言，逻辑混合则要复杂很多。

注 13：我们下周要去健身房，因为听说新开了一个旋转课程，想去看看。

我们知道，在线性混合中，没有反馈回路来说明在预测如何压缩数据时我们赋给一个模型的权重是否正确。因此，当输入数据流变化时，模型的权重保持不变，最终得到的结果不但没有压缩，反而比原来需要的空间还大。

为了解决这个问题，逻辑混合使用了神经网络（你没有看错，就是人工智能中的神经网络）来更新权重，而更新的依据则是哪个模型在过去给出了最准确的预测。这里的关键是修改当前的权重。假定根据当前的权重，我们选择了模型 A 编码某个符号，编码后的长度为 12 个二进制位，而模型 B 其实才是正确的选择，它只需要 8 个二进制位。那么，编码器会输出 12 个二进制位，并修改模型的权重，这样当下次所有的模型输出结果相同时，模型 B 有更大的机会被选中。

逻辑混合的缺点是，在进行数据压缩时，它需要消耗大量的内存，同时运行的时间也较长。如果看一些在 LTCB 上运行的 ZPAQ，你会发现压缩 1 G 的文件需要 14 G 的内存。这些压缩的中间数据都需要空间去存储。

9.4 下一代技术

上下文混合算法给数据压缩的未来指明了方向。总的来说，如果对所需的内存与运行的时间不加限制，同时还有足够的数据建模知识，那么最优压缩就是个已解决的问题。这有可能是云计算层次上数据压缩的下一个大的解决方案。对于那些拥有大量计算资源和时间的公司来说，**只要它们的数据科学小组正确地理解了所有模型**，它们就能大规模地进行数据压缩了。

然而这种情况目前还没有在消费市场上出现。由于需要大量的内存和运行时间，这就使得上下文混合算法很难适用于移动设备（至少在本书写作时如此）。然而实际情况是，这时的数据目标与前面的完全不同。如果你想处理的只是 1~50 MB 的数据，那么即使用了上下文混合算法，所得的结果也会和很多其他的算法差不多。只有当需要处理的数据量很大、数据很复杂并且一直在变化时，上下文混合算法才能真正发挥作用。

因此，我们再次得出了这样的道理：对数据压缩来说，同样没有银弹。无论是哪个数据集，都需要有思考的过程，对信息的定义和处理进行分析。即使是上下文建模，虽然它建立的目的是适应数据的变化，但也同样依赖于人们所建的信息模型。

这就意味着一件事是确定的，即数据压缩还远不是一个已经解决了的问题，在这个领域内，还有很多让人惊叹的事情等待着我们去发现。

第10章

换个话题

到目前为止，本书主要关注特定的数据压缩算法以及它们通常是怎样工作的。虽然这些内容中包含的信息很多，但除非你打算自己实现一个有突破性的数据压缩工具，否则这些内容的主要目的还是帮助你理解数据及其压缩。现在，我们想换个话题，谈谈数据压缩应用中的一些点，以及它们如何与你、你开发的项目以及眼前的世界关联。

当前压缩可以分为两类，即**多媒体数据压缩**（media-specific compression）与**通用压缩**（general purpose compression），下面来分别讨论。

10.1 多媒体数据压缩

多媒体数据压缩工具是专门设计用来压缩图像、音频、视频等多媒体数据的。一般来说，你的应用程序发送、接收、处理、存储并且向用户展示得最多的就是这些类型的内容。俗话说"一图胜千言"，用到数据压缩上，同样正确。一张 1024 × 1024 的 RGB 色彩模式的图片，其大小就有 3 MB，如果用 ASCII 码来表示字母的话，同样的空间能用来表示 3 145 728 个字母。相比而言，有名的《霍比特人》一书只有 95 022 个单词，如果假定平均每个单词由 5 个字母组成，那么这本书大约有 475 110 个字母。也就是说，一张 1024 × 1024 的图片所占用的空间，可以用来存放约 6 本《霍比特人》这样篇幅的书。

这也是为什么大多数多媒体数据压缩工具使用**有损**压缩算法。有损压缩指的是为了使数据压缩得更小，可以牺牲多媒体的质量这样的数据转换。例如，一张 1024 × 1024 的图片，如果使用 8 个二进制位来表示红绿蓝这 3 种颜色通道，那么每个像素就需要 24 个二进制位，因此整个图片的大小为 3 MB。然而，如果每个颜色通道只用 4 个二进制位表示，那

么每个像素就只需要 12 个二进制位，整个图片占用的空间也就只有 1.5 MB，但同时也降低了图片的色彩质量。[1]

有损数据转换的种类特别多，每一种都针对特定的多媒体文件（针对图像文件的就不太适用于音频文件）和内容类型（灰度图像与全彩图像使用的压缩算法同样不同）。除了这些转换是有损的外，前面讨论的内容同样适用于有损压缩工具。在内容转换为更容易压缩的状态后，你可以继续应用所有标准转换，例如 LZ、BWT、RLE 以及增量压缩，甚至可以使用哈夫曼编码、算术编码和 ANS。关键在于，针对某种数据类型找出最佳的转换方法，以获得最好的结果。

本书没有花太多的时间来讨论有损转换，这是有意为之的，因为这样的转换实在太多，并且每一种都针对特定的内容，可以说每种多媒体类型都需要用一本书的篇幅去讨论。[2] 不过不用担心，第 12 章会介绍一些重要的图片压缩优化的细节，不过没有太过深入。

10.2 通用压缩

相比而言，通用压缩工具是设计用来压缩除多媒体数据以外的其他数据。像 DEFLATE、GZIP、BZIP2、LZMA 和 PAQ 这些算法，都是将各种无损转换结合起来，用来压缩诸如文本、源代码、序列化数据以及二进制内容等其他不能使用有损压缩工具压缩的非多媒体文件。这方面的研究同样也很多。顺便说一下大文本压缩基准测试，有很多通用压缩工具会参与这项压缩大文本的比赛，以比较这些工具的各种性能指标。新的算法还在不断出现。谷歌对 GZIP 算法的改进已产生了一系列的压缩工具，如 Snappy、Zopfli、Gipfeli 以及 Brotli[3]，这些工具努力的方向是为了实现更好的压缩率、更小的内存需求和更快的解压速度，但是每个的侧重点不同。

你每天下载的大部分互联网内容是用这些算法中的一种压缩的。标准的 HTTP 协议栈允许数据包使用 GZIP 和 BZIP 编码，现在又多了一种 Brotli（前提是服务器端和客户端都支持），也就是说，网页、JavaScript 文件、留言以及商店列表这些内容都会在解压后显示到你的设备上。

值得指出的是，很多数据压缩研究人员，包括我们两位作者，认为所有这些算法都陷入了回报率递减的困境。不妨看看最近那些很成功的压缩工具（Brotli、LZHAM、LZFSE、ZSTD），我们发现有以下趋势：研究主题的变化很小。它们都是在现有的转换上使用一些

注 1：将颜色通道从 8 个二进制位减为 4 个二进制位，我们就将能表示的颜色数量从 1600 多万种减为 4096 种，而人类的眼睛能识别的颜色为 100 多万种。想了解得更多，可以查看维基百科词条“三原色”（Trichromacy）。

注 2：是不是不太相信？没有关系，你可以先了解一下 JPG 格式的工作原理，或者复习一下最新的 WebM 格式。

注 3：Gipfeli、Zopfli 和 Brotli 的命名都来自于瑞士面包，或许，可以将谷歌称为“一家算法面包店”。

技巧或是进行一些改变，然后再应用到现有的压缩算法上，以获得某些小的提升，而且获得边际提升需要的资源越来越多。通过观察各种各样的基准测试，我们发现 30%~50% 这样比例的突破已经不存在了，更多的是经过大量努力后，只能在现有算法的基础上提高 2%~10%。

10.3 实践中的数据压缩

我们希望你在学习本书介绍的这些知识后，能很容易地将它们应用到你开发的应用程序中去。还有一些内容我们特别想介绍，这些内容可以帮助你更多地了解如何在应用程序的开发中实现那些比较容易达到的目标。接下来的几章将重点放在理解如何评估数据压缩的各种指标上，并给你一些关于图像数据和序列化数据的建议，最后介绍我们对未来 10 年中数据压缩重要性的宏观思考。

祝你学习之旅愉快！

第11章

评价数据压缩

在让压缩影响到你那不错的应用程序的每个部分之前，一定要注意其中涉及的权衡取舍及其使用场景。并不是每个算法都适用于所有的使用场景，在某些情况下，同一压缩格式的不同实现可能会更好地满足你的需求。

因此，当谈到数据压缩时，什么最重要呢？

11.1　数据压缩的使用场景

我们从以下关注点开始讨论，即数据是在哪里压缩、存储和解压的。理解数据是从哪里来的、到哪里去之所以很重要，是因为编码器与解码器之间的相互作用很重要，这一点后面会有更多的讨论。首先，我们来看一下 4 种常见的使用场景。

11.1.1　线下压缩，客户端解压

在第一个场景中，数据在某个与客户端无关的地方压缩，然后分发到客户端，并在客户端解压以供使用。

这一场景对打包的应用程序或者电子游戏来说很常见，并且相应的资源文件中常常包含大量的图片、视频和音乐。另一个场景就是艺术家创作并共享他们的作品。无论是哪种情况，原始的作品都是使用高分辨率、高保真的工具制作的，然后再输出并压缩以供分发。

这里压缩的目的是使多媒体文件尽可能地小。

需要权衡取舍的是多媒体文件的品质。

11.1.2 客户端压缩，云端解压

大多数现代社交媒体应用程序会在客户端产生很多内容，然后将它们推到云端进行处理并分发给其他社交用户。在这些场景中，通常会在客户端进行初步压缩，以节省出站通信的流量费用。例如，在获取一组社交数据后，我们会先使用二进制序列化格式将这组数据序列化，然后在发送到服务器之前再用 GZIP 对它进行压缩。

这里的主要目的是减少用户的费用。虽然很多北美的用户（仍然）使用的是无限量数据套餐服务[1]，但是世界上很多地方的用户不能享受这样的服务，因此大多数用户使用的是按照流量付费的套餐。也就是说，这些用户上传到服务器上的每个二进制位都要自己付费。

这里的权衡取舍是，对于移动设备，需要消耗电池的电量去压缩数据。

11.1.3 云端压缩，客户端解压

放到云端压缩的数据大致可以分类两类，这两类的特点完全不同，下面分别叙述。

1. 由云端资源生成的动态数据

与用户将数据上传到云端相反，这些数据来源于云端并由用户下载到本地。

例如，在客户端请求数据库操作的结果，或者服务器发送了动态布局的数据，而客户端在等待内容生成的时候，服务器生成并压缩数据所需的时间就变得非常重要。因为如果需要的时间比较长的话，客户端就会明显地感受到网络延迟。

这里最重要的是平衡压缩后的大小与所需要的时间。值得指出的是，在某些高延迟的环境下，用户可能会更愿意多等一些时间，使文件变得更小。

这里，压缩的目的就是让通过网络传输的内容变得更小。

这里需要权衡取舍的是时间。

2. 为提高计算效率而传输到云端的大量数据

这一场景的重要性往往是因为需要确保手边的媒体文件尽量地小。例如，设想你有 2 GB 的 PNG 文件需要转换为 10 种不同分辨率的 WebP 图片，或者是你有长度为 1200 个小时的视频文件需要在播放前转换为 H.264 格式。

需要记住的是，由云端传输出去的每个二进制位都需要所有者付费，实际上，客户端也需要为从云端获取的每个二进制位付费。因此，这是使用云端计算资源的理想场景，我们可以用计算效率最高的方式让数据的二进制位数变得最小。

注 1：在我们写作本书的时候，这种状况也正在发生变化。

这里的目标就是高效地将大量的数据压缩为最少的二进制位数。

这里需要权衡取舍的是成本和效率（也就是计算资源的价格）。

11.1.4　客户端压缩，客户端解压

最后，还有很多客户端应用程序相互之间需要通信，它们之间可能会相互发送对等网络包、图片或者 GPS 信息这样的内容。

在这些情况下，客户端会生成数据并压缩它，然后再发送给其他客户端去解压。

这里的难点是，客户端通常是移动设备，没有优化转换和压缩数据所需要的大量资源。然而，这些设备通常会有专用的图形硬件，因此可以用来压缩像 JPG 和 H.264 这样格式的文件，也就是可以用来处理图片和视频。至于其他类型的数据，因为只有这样的硬件配置，所以数据压缩的质量会比较低（没有什么时间去优化），同时解压缩的过程也会比较慢（移动设备的电量有限）。

因此，处理客户端之间互相通信的算法通常选择固定二进制位率的压缩，例如手动打包序列化结构而不是对它进行压缩。

如此一来，这里就不存在取舍，相反，这里更多的是需要权衡设备的功能、压缩和解压需要的时间以及需要数据的迫切性。

11.2　数据压缩的需求

前面在介绍不同压缩算法时提示过，理解不是所有的压缩算法和格式都适用于所有类型的数据，这很重要。例如，如果对图像数据应用哈夫曼编码，那么压缩的结果肯定达不到应用有损图像压缩算法[2]的水平。

作为开发人员，为每种类型的数据匹配正确的算法，对最大限度地压缩数据至关重要，当然在这中间需要做出一些权衡取舍。永远不会有银弹，做出正确的选择需要我们：

- 了解要处理的数据——要了解的不仅是数据的类型，还要了解它的内部结构，特别是它的使用方式；
- 了解算法的各项指标，这样才能从中选出正确的算法系列；[3]
- 最重要的是，了解在给定的情况下你需要的是什么，因为有些算法能节省特别多的空间。

那么在实践中这意味着什么呢？

注 2：即在压缩的过程中会丢失部分信息的算法，关于有损压缩的知识，参见第 14 章。

注 3：更具体地说，适用于大量文本的算法可能在压缩数值型数据时效果并不好，反之亦然。

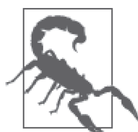
举个例子，与全屏显示的图片相比，缩略图对图片质量的要求就会比较低。因此，缩略图可以使用有损的 JPEG 编码压缩，而对质量有更高要求的图片则应该使用无损的 WebP 编解码器编码。

11.3 压缩率

压缩率，也就是内容压缩后的大小与压缩前大小之比，通常是评价压缩效果时最重要的指标。因为压缩的最主要目的就是让数据变得最紧凑，在网络传输中二进制位数总是越少越好。

当然，总是会有一些例外。当性能或者内存更重要的时候，你可能愿意接受压缩节省的空间变小。这里就有一个很好的例子：1 GB 的文本文件，用 ZPAQ 算法压缩，压缩后的文件通常会最小，但同时需要 2 GB 的内存以及 3 个小时才能在台式计算机上完成压缩，解压时需要的资源时间大致相同。因此，当关注的主要是压缩后的文件大小时，ZPAQ 算法是很不错，但它不适用于在移动设备上压缩数据。

当然，对那些在线下或者云端进行压缩的服务来说，压缩率就是最重要的考虑因素之一，这里我们有资源、有时间将数据压缩得最小，同时这样做还能减少传输数据所需的费用。



用户总是处于劣势

值得指出的是，用户总是处于天平劣势的一端。世界上的大部分地区没有“无限量数据”套餐，而且用户下载 1 GB 数据的费用与服务器提供 1 GB 数据的费用差别极大。

如果想做一个用户愿意使用的移动应用程序，那么你应该站在用户角度去考虑数据传输的费用。

11.4 压缩性能

压缩性能，指的是将数据转换为压缩后的形式需要多长时间。在对网络延迟要求很高的情况下，无论是客户端负责数据压缩还是客户端在等待服务器正在压缩的数据，压缩性能都至关重要。

在这个方面通常有两个评价指标：CPU 速度和内存。编码系统的 CPU 速度之所以重要，是因为它决定了数据可以压缩得多快。而可用内存的数量之所以重要，是因为它十分有限，特别是对移动设备来说。

举个例子，LZMA 算法的压缩结果让人印象深刻，但同时该算法也需要大量的内存。这就使得它对移动设备来说没有太大的吸引力，因为移动设备可能只有 256 MB 的内存。

大多数客户端（至少是移动设备）内置了对硬件压缩编解码器的支持，至少是对某些有损压缩类型的数据。像 JPG 和 H.264 这样的文件可以很容易传输到这些硬件上，而服务器端通常没有配置这些特殊的硬件，因此，在客户端进行压缩要比在服务器端更容易。

对需要无损压缩的数据类型，我们甚至看到了一些 GZIP 芯片。由于这些硬件组件都是专门设计的，因此与软件实现相比，它们的速度要快很多，只要有机会就应该去使用它们。需要记住的是，在客户端压缩能使用的资源相当有限，如果能改变策略优化性能，肯定有好处。特别是遇到序列化的数据需要通过对等网络传输的情况，对每帧数据包及其位置的更新都会导致数据的加载，因此利用 GZIP 硬件的好处更多。

11.5　解压性能

对所有重点关注性能的环境来说，解压速度的重要性超过其他所有指标。在现代应用程序的开发中，解压通常是在客户端设备中进行，与服务器端相比，客户端通常存在能量不足的问题。那些能将文件压缩得最小的算法，通常也需要花最长的时间去解压，因此对需要将数据传输到移动设备上处理的应用程序来说，这些算法并不适用。

因此，有时我们必须做出取舍：选择压缩算法主要是根据该算法的解压性能而不是压缩后文件的大小。例如，ZPAQ 可以说是一种非常高效的压缩算法，它使用神经网络作为编解码器，因此解压时需要运行大量的资源。这样的资源需求使得它注定与只拥有较少的 CPU 和电池资源的移动设备无缘。

WebP 图片格式是另一个很好的例子。最初几个版本的 WebP 相比 JPG 来说，所需的空间更小但是图片质量更高，美中不足的是解码所需的时间几乎是 JPG 的两倍。因此，很多公司对于是否采用这种格式很犹豫，而在随后的版本中，WebP 解码的性能提高了，最终很多公司采用了这种格式。

在台式设备和移动设备中普遍存在的硬件解码器，正在改善这种情况。处理 JPG、OGG 和 H.264 的硬件芯片，正在提高专门为它们而设计的算法的解码性能，让它们成为某些情况下更好的选择。

实际上，GZIP 之所以成为当前世界上使用较多的通用文档压缩算法，解码性能是其中最主要的原因之一。GZIP 算法生成的压缩文件大小合适且解压速度很快，这使得它适用于各种类型的嵌入式设备和非嵌入式设备。从诞生到现在的 20 多年里，GZIP 算法一直在改进，其解压性能不断提高到新水平。

11.6　解码流的能力

数据流通常是解压时容易被忽略的一个方面。我们通常认为解压算法处理的是“完整的数

据包”，也就是说解码前所有的数据都必须在内存中。

然而这样的想法远非事实。想象一下，一次去听一部歌剧或者观看 Kenneth Branagh 的《哈姆雷特》，通常你会认为这样做一下子接受的内容太多了，最好能一次只听一两个章节或只看一两幕。

幸运的是，一些通用压缩算法如 GZIP、BZIP2 以及大多数多媒体压缩算法像 H.264 能在流模式下工作。数据以分块的形式发送到客户端，一到客户端就开始解码（即分块解码）。对许多客户端应用程序来说，这种能力正是它们需要的。想象一下，一位用户想要从头浏览某个社交视频流，而在他能解码“我终于在巨石阵趴街了”这一内容前，必须要先去下载过去 10 年里“今天早晨我准备吃什么”这样的内容，这真是令人难以接受。

11.7 比较压缩算法

由于有这么多压缩格式和算法，因此有时候就难免想对它们进行逐一比较，看看对于给定的某种数据类型哪种算法表现最好。

幸运的是，这样的工作不需要你自己去做。例如，前面提到的大文本压缩基准测试会定期比较各种算法压缩 1 GB 文本数据时的各方面表现。Squash 压缩基准测试则会测试各种算法在压缩 XML、文本、图像以及其他数据格式时的表现。此外 Squeeze Chart 则会比较算法在压缩各种文本、音频以及位图时的表现。

总之一句话，不同的算法和不同的设置，都会影响到开发的应用程序的压缩质量。在任何情况下，你都需要根据当前的条件和目标，试着对候选的算法进行测试，从而挑出最适合的那一个。

魏斯曼评分

我必须要感谢 Mike Judge，他帮助压缩领域摆脱了困境。20 世纪 80 年代以来，数据压缩领域的成果相对较少且进步缓慢。是的，在 20 世纪 90 年代出现了 BWT，在 21 世纪初出现了 LZMA，到 21 世纪 10 年代又有了 ZPAQ，可是除此之外，就再也没有能拿出手的东西了。然而，在一阵笑声中，突然整个世界再次对压缩有了兴趣。到底发生了什么？

2014 年，一部名为《硅谷》的电视剧上映后大受欢迎。这部电视剧展示了一位程序员的创业之旅，剧中主人公试图创建一家颠覆性的压缩算法公司。虽然这部作品的核心是讽刺，但它对压缩领域的影响很大。突然，关注数据压缩变得很酷。媒体也沉迷于这样的故事，任何关于某个公司正在做一些与压缩有关的有趣事情的故事，都会被立即拿来与这部电视剧比较。因此，当谷歌发布一种新的压缩算法时，媒体就借此机会讨论艺术模仿生活，生活模仿艺术之类的话题。

虽然这部电视剧中的压缩算法都是虚构的，但是制片人还是想从中展示一丝真理，所以他们联系了斯坦福大学的教授塔奇·魏斯曼（Tsachy Weissman）帮助他们达成这一目标。因此，魏斯曼教授设计了一种度量数据压缩性能的方法，用数据集的压缩率除以其编码速度作为衡量标准。其目的是通过已知的现有编码器（如 GZIP），归一化新压缩算法的压缩率与其编码速度之商，来测试新算法的性能。通过归一化，我们就有了将某个算法与通用标准压缩工具相比较的能力，这有助于评估对于某种数据类型，哪个算法最合适。

制片人以它的提出者为这种新的度量标准命名，称之为魏斯曼评分（Weissman Score），从此之后它成为互联网上的传奇，尽管并不清楚这一评分在压缩的实际应用中是否有根据。（至少在 encode 这一数据压缩论坛上，目前还没有人使用这一评分……）

这里值得一提的是，由于人们对数据压缩领域有了更多的关注，我们希望见到专注于这一领域的新一代算法和研究，从而帮助数据压缩取得新的突破。

第 12 章

压缩图像数据

如果你是一名应用程序开发人员，那么很有可能你的应用程序的绝大部分内容是图像数据。社交流媒体、购物网页，甚至是地图信息，所有这些图像内容都必须**不断地**发送给用户。

图像压缩是一个非常棘手的问题，包含在压缩工具中的每个有损压缩算法都很复杂（最乐观地看），还是不要接触的好。不过也不要放弃希望，虽然这些压缩工具大多数情况下被当作黑盒系统，你的开发团队仍然可以做很多事情来影响图像内容的大小，使它变得更小。

12.1　理解图像质量与文件大小

通常来说，图像压缩工具会提供一个整数参数，让你来决定图像的质量。[1]

随着这个值变小，图像的大小也会变小，当然质量也会变差。

你知道，这个值主要是用来控制有损算法为了压缩效果更好，而在转换数据时采取的力度。更差的质量就意味着有更多的颜色被丢弃，或者是有更多的边缘信息被忽略，所有这些都是为随后的统计编码阶段生成更多的重复符号。

选定这样一个值，可以说是一项重大的、关键的、耗时的决定。如果这个值选得过小，就会造成图像伪影，用户也可能抱怨图像质量差；如果这个值选得过大，那么发送的图像就比需要的大得多，相应的费用也会增加不少。图 12-1[2] 展示了这一过程。

注 1：或许你对这个概念有所了解，当在 Photoshop 软件中保存 JPG 文件时，它通常会询问你想要什么样的“质量等级”。

注 2：图 12-1 的彩色图片，请通过本书图灵社区页面（https://www.ituring.com.cn/book/1893）的“随书下载”处获取。——编者注

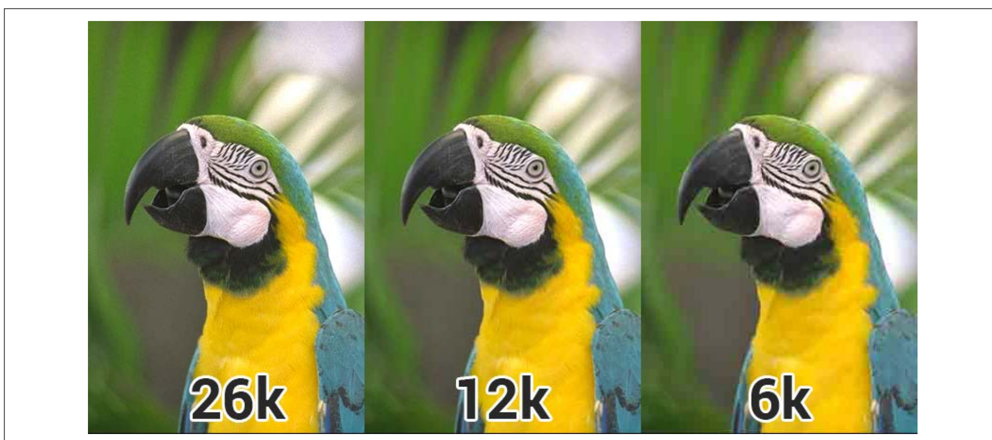


图 12-1：一组有着不同压缩率不同质量的图片，图片质量从左到右依次为高、中、低。随着图片变得越来越小，其质量也变得越来越差

对少量的图像，艺术家或者设计师可以手工选择最佳的质量值。在使用工具导出图像时，他们可以通过滑动质量值条，找出图像质量和最终文件大小之间的最佳平衡。

然而这样的方法不能规模化。如果你有 1500 万用户都将他们做的美食照片上传到服务器，而你却做不到去雇一批艺术家找出最佳平衡点。更糟糕的是，每张照片的最佳平衡点都可能不同。一个真正平滑的图标或者一张落日的照片对质量的要求，与一张森林或者脸庞的照片对质量的要求是不同的。人脑构造就是这么复杂，在不同的场景中注意的图像质量问题不同。

因此，这就提出了一个价值百万美元的问题：在规模庞大的情况下，怎样才能找出每幅图像的最佳质量值？

遗憾的是，今天大多数的开发人员没有试着去解决这个问题，而是最终只选定一个质量值并将其应用到服务中的所有图像上。

正如 imgmin 开源项目指出的那样，对于级别在 75~100 的 JPG 压缩，通常用户只能感受到很小的质量差别。

> 对正常的 JPEG 图片来说，当压缩级别在 75~100 时，只会出现非常小的、几乎不可见的“明显”质量变化，但文件大小之间的差别比较大。这意味着当质量值为 75 时对普通用户来说很多图片看着挺好，但是其文件大小只有质量值为 95 时的一半。当质量值低于 75 时，图片看起来就变差很多，并且节省的空间也在逐渐递减。

imgmin 开源项目还进一步指出，大多数的大型网站往往将所有 JPG 图片的质量值设定在 75 左右，具体数据如下表所示：

公司图像类型	JPG质量值
谷歌图片缩略图	74~76
Facebook 原尺寸图片	85
Yahoo 首页 JPG 图片	69~91
YouTube 首页 JPG 图片	70~82
维基百科图片	80
Windows Live 背景图片	82
Twitter 用户 JPG 图片	30~100

这里的问题是，这些选定的值不太理想。它们通常是在脱离实际的环境中选定的一个值，然后被应用到了整个系统中的所有图像上。实际上，有些图像可以进一步压缩，其质量的损失可以忽略不计，而另一些图像则因为压缩得太多而看起来不太好。

12.1.1 是什么降低了图像的质量

在看图像时，人类的眼睛对很多事物很敏感，其中包括边缘和渐变。[3]

任何时候，当两个已知值之间的边缘出错，或者出现了与大脑认为的平滑颜色不一致的搭配，都会很容易察觉（见图 12-2）[4]。

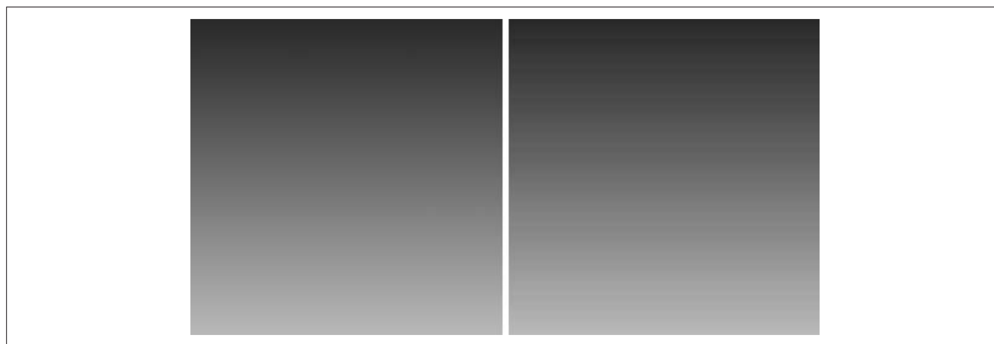


图 12-2：完全平滑的渐变图（左）与压缩后的渐变图（右）。左边的源图片有近 128 种不同的颜色，而压缩后的图片则只有 32 种。这种由于颜色的缺失而造成的现象通常称为“色彩断层”（banding），它降低了渐变图像的平滑质量

量化（quantization）和**区块化**（blocking）是导致图像压缩出现视觉问题的最常见的两种形式。大多数图像压缩算法将图像数据先切分为像素块，然后再量化，以减少图像中不同

注 3：看一张图片时，通过直觉我们就能感受到画的质量好不好。因此，虽然“质量”这样的表达可能有些含糊，但它其实包含了颜色准确性、清晰度、对比度和失真度这些概念。而我们真正想要的是度量质量的能力，这一内容随后会展开。

注 4：图 12-2 的电子图片，请通过本书图灵社区页面（https://www.ituring.com.cn/book/1893）的“随书下载”处获取。——编者注

的颜色数量，再在此基础上基于图像的局部性进行修改。

例如，JPG 会将图像的像素切分为 8×8 的小块，然后试着找出与此区域相似的颜色。这种方法之所以可行，是因为图像数据局部区域之间存在关联。也就是说，在一幅真正随机的图像中，两个相邻的像素之间并不会存在相关性；然而在一张照片中，两个相邻的像素之间往往是渐变的，颜色相似。这种区块化过程造成的结果是，相邻区块之间的颜色可能不太相同，区块之间的边缘因此也就变得很明显，如图 12-3 所示。

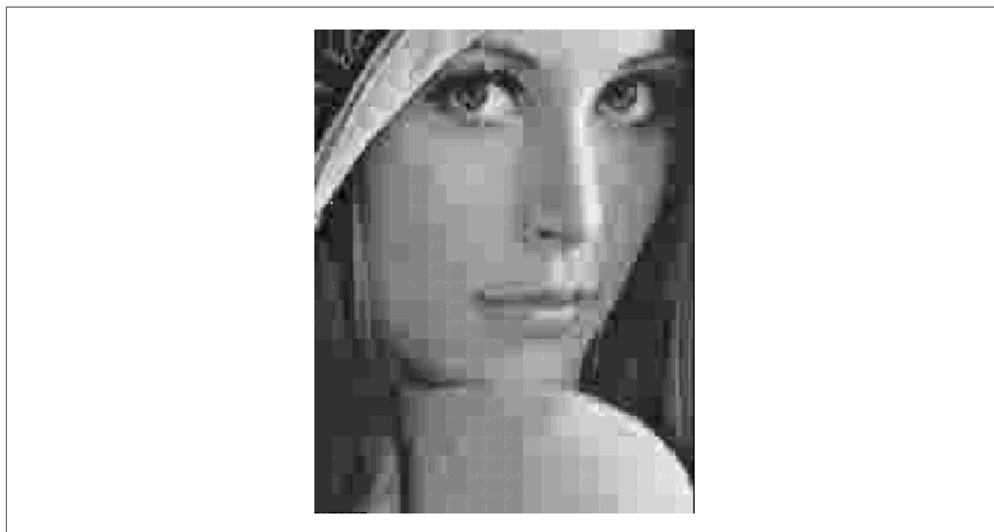


图 12-3：用于展示区块效应影响的莱娜图特写

事实上，当前已有很多研究致力于弄清楚由于压缩而导致的不同类型的视觉伪影（visual artifacts）。

12.1.2 度量图像质量

人脑虽然可以本能地觉察到并界定出哪些图像质量很糟糕，但在需要自动判断经压缩处理后图像质量“好不好”时，它不起太大作用。因此，提出一种数学上的、可度量的，因而能用编程实现的“图像质量”概念就变得尤为重要。

如今，人们通常使用相互之间有些竞争的两个指标来评价图像数据：**峰值信噪比**（peak signal-to-noise ratio，PSNR）和**结构相似性**（structural similarity index，SSIM）。

PSNR 通常表示一个信号的最大可能功率与影响它的表示精度的破坏性噪声功率的比值（以对数分贝为单位）。这一度量的基础是压缩图片的均方误差（mean-square error，MSE），换句话说，原始图像的值与压缩后的值差别有多大。

PSNR 与 MSE 之间，存在着反比关系。当误差的数量较少时，图片的质量就比较高（PSNR 同样如此），反之亦然。这里唯一需要注意的是，如果你试着去计算两张一样的图像的均方误差值，其结果会为 0，那么对应的 PSNR 会是未定义的（除以 0）。

然而 PSNR 的度量还是存在一些问题的。因为它计算的是去噪之后经过均方处理的误差，所以它稍微有些偏向过度平滑（即模糊）的结果。用通俗的话来说，即使将图像的部分结构移除了，得分还是比较高（忽略了图像缺失的那部分）。因此，PSNR 并不总是与源图像或者应用到其上的效果类型相一致。此外，PSNR 严格依赖于数值比较，没有考虑任何与人类视觉系统相关的生物因素，因此，就会出现从数值上去看很好，但用眼睛去看就有比较明显的图像质量问题的情况。

SSIM 这个概念的提出就是为了解决 PSNR 的问题，在比较图像的压缩质量时考虑了人眼的感知情况。它是通过比较源图像与压缩后图像的边缘相似性来实现的。SSIM 看上去是一个更好的质量度量标准，但是其计算也更复杂。

SSIM 取值范围为 [0,1]，当其取值为 1 时表示的是压缩后的图像与源图像完全相同，而取值为 0 则表示压缩后的图像与源图像完全不同。

图 12-4 展示了一系列照片中每张照片的 MSE（PSNR）和 SSIM 的值。

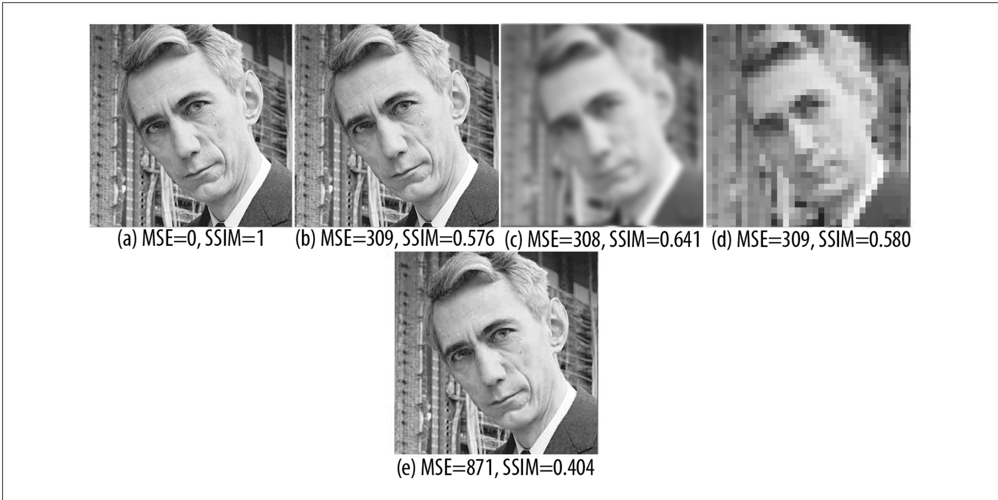


图 12-4：具有不同 MSE（PSNR）和 SSIM 值的图像并排显示，其中 (a) 是原始图像，(b) 增加了一些白色噪声，(c) 增加了模糊效果，(d) 是低质量 JPG 压缩后的结果，(e) 则是高质量 JPG 压缩后的结果。值得注意的是，虽然图像（b）、（c）和（d）的 MSE 值相差无几，但其给人的视觉效果完全不同

12.1.3 让想法真正工作

显然，为了让用户感受到相同的视觉品质，不同类型的图像需要有不同的输出设定。一张森林的照片由于有很多边缘和不同的颜色，因此需要更多的二进制位以充分地展示它；而一张手绘的卡通，因为其颜色简单，所以不会产生很多的渐变问题。因此，不同的图像类型，需要使用不同的输出设定。然而，如何在开发环境中实现它，这个问题留给你来解决。

显而易见，通过云端的计算资源反复寻找找出理想的质量设定是最简单的方法，但并不是所有的开发人员都有时间和经费做到这一点。

12.2 图像的尺寸很重要

在现今的移动世界，有很多屏幕尺寸不同、处理能力各异的设备。这对开发人员来说是一个很大问题：在处理图像时应该使用什么样的分辨率？

考虑这样一个场景，一位用户正在用手机上传一张 800 万像素的照片。[5]

对那些屏幕分辨率与上传用户相同的用户来说，有理由相信他们希望看到的是同样大小的照片，可是应该如何应对那些屏幕分辨率只有一半或者四分之一的用户呢？这与从桌面显示器上和手机屏幕上看图像时的差别类似。显然，更小的屏幕会使用更小的物理像素显示图像，因为它拥有的物理像素就少，那么应该在什么地方调整大小呢？

将全分辨率的图像发送到设备上，在渲染前再调整大小，对开发人员来说，这样做肯定最省事，但缺点也很明显，我们将用户不需要（也不会看）的数据发给了他们。这样做，其实是把钱往水里扔，还听不到一个响。

更好的方法是直接在云端调整图像的大小，或者在某处缓存调整大小后的图像，这样就能向小屏幕发送小尺寸的图像。这并非遥不可及。很有可能，我们已经有了同一幅图像的不同分辨率的多个版本（见图 12-5）：低分辨率的缩略图、高分辨率的全屏版本，也许还有介于两者之间的预览版本。我们可以在云端或者本地使用自动化工具，只要调用一次就能生成所需的所有分辨率大小，无须艺术家在图像编辑器中手工生成。

注 5：有可能拍的照片就是中午吃的克林贡血肠。

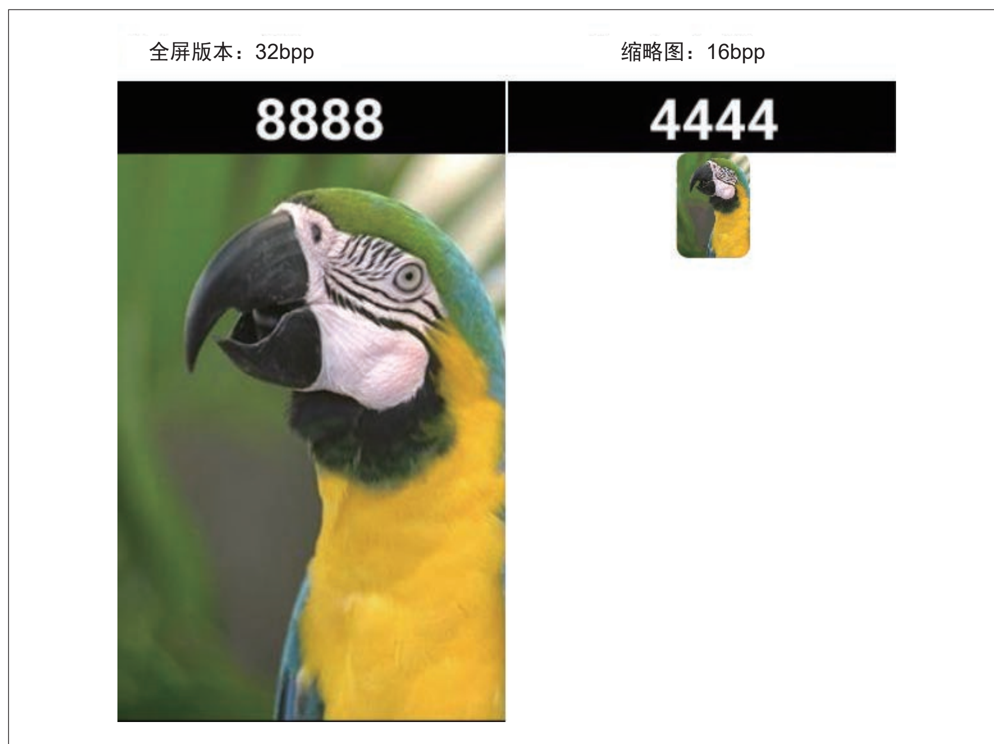


图 12-5：图片尺寸也为修改图片质量提供了方便。全屏的图像由于有更多的视觉空间，因此用户容易发现图像质量问题。像缩略图这样比较小的图像，我们就可以接受更多的质量格式问题，因为用户可能不会注意到或者去抱怨

发送合适大小的图像给用户有以下好处。

- 发送的数据量更少了，这会更快，也会节省用户的套餐费用。
- 可以节省用户的设备空间。
- 无须再调整图像的大小。（是的，我们知道 GPU 可以帮我们去做这件事，但是如果要在 GPU 中处理 4 MB 大小的图像，而渲染的时候只是缩略图，是不是很浪费资源？）
- 解码会更快，加载会更快，显示也会更快……你都明白了吧。

12.3 选择正确的图像格式

前面提到过，目前的图像压缩算法和格式有很多，每一种的权衡取舍都不同，适用场景也不同，这一点我们在开发图像压缩管道时应该注意。下面讨论今天在移动应用程序和网页开发中最常用的几种图像格式：PNG、JPG、GIF 和 WebP。

12.3.1 PNG

便携式网络图像格式（Portable Network Graphics format，PNG），是一种无损图像格式，它使用 GZIP 这样的压缩工具使数据量变小。由于是一种无损的图像格式，因此压缩后的图像质量与源图像相同。这正是它的优点，既能保证图像的高质量又能压缩数据量，当然压缩的程度不像有损压缩那么大。

PNG 格式最吸引人的地方在于它对透明度的支持。除了红、绿和蓝这 3 种颜色通道外，它还支持 alpha 通道，可以定义渲染时哪些像素是透明混合的。对透明度的支持，使 PNG 对网站、对那些希望屏幕上的图像不是矩形的应用程序有很大的吸引力。当然，多了一个 alpha 通道是有代价的，那就是文件的大小也变大了。

PNG 格式还允许文件中存在元数据块，这使得图像编辑器（以及生成图像的客户端设备）可以将额外的数据附加到文件中。虽然这是有用的，但通常它也是数据膨胀的主要原因，而且大多数情况下，这些描述图像是由什么程序生成的数据，对用户来说其实是垃圾数据。因此，在发送图像前，去掉这些用户不需要的数据很重要。[6]

事实上，PNG 格式的这种无损性质是一把双刃剑。从图像质量的角度来看，相对源图像我们可以获得完美的像素结果，但是从文件大小的角度来看，复杂的图像不会包含很多颜色相似的像素，因此压缩就会不太理想。如果真的需要使用 PNG 格式，比如 Android 应用程序打包或是网页上需要使用透明的图像这样的场景，我们可以在图像保存为 PNG 格式**之前**，进行一些有损的预处理，来提高图像的压缩率。

幸运的是，那些让人抓狂的有损预处理的代码不需要你亲自去写，有很多应用程序可以帮你去做。只需要在网上搜“有损 PNG 压缩工具”，就会出现很多此类的工具。至于具体选择哪一种预处理器，则主要取决于你的需求以及你要处理的数据集的性质。

12.3.2 JPG

如果你对透明度没有明确的需求，那么联合图像专家小组（Joint Photographic Experts Group，JPEG 或 JPG）格式可以说是一个更好的选择。JPG 是一种用于摄影图像的格式，它不支持 alpha 透明度。它包含一个功能强大的有损压缩工具，我们可以通过一个质量值来控制它，以达成对文件大小与图像质量的权衡取舍。

JPG 这种压缩格式的基础是分块编码。如图 12-6 所示，一幅图像会被分成 8×8 的小块，然后在每块上应用各种不同的变换，再将变换之后的小块组合起来交给统计编码算法处理。

注 6：大多数能导出 PNG 格式的图片编辑程序能去掉这些信息，很多专门的 PNG 压缩工具也有这样的功能。

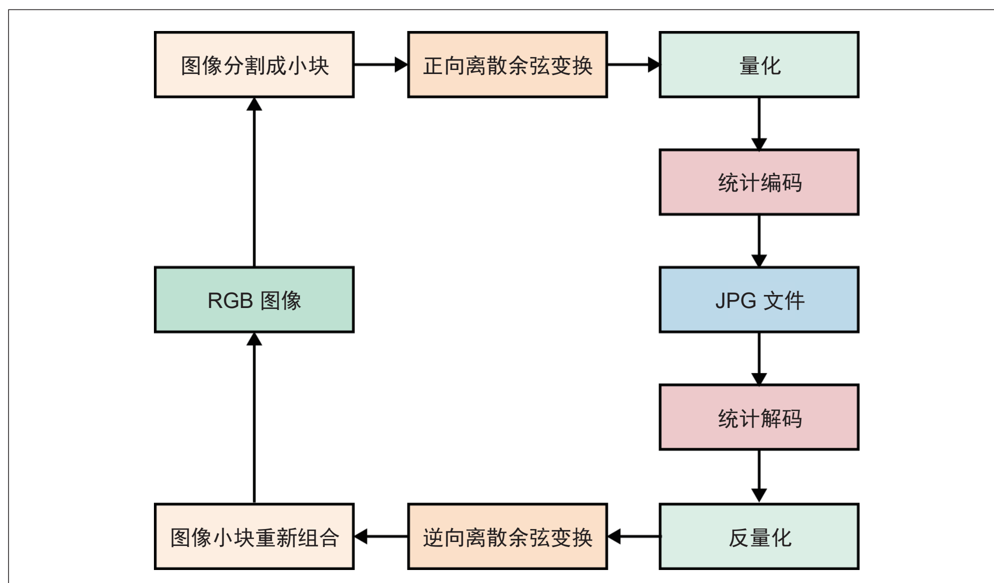


图 12-6：JPG 算法的工作原理图



分块只对照片适用

需要注意的是，分块过程只对摄影图像（即照片）适用。如果你想压缩的是颜色比较少的图像，比如手绘的卡通画，那么 PNG 对应的无损压缩工具表现得会更好，因为它可以将很长的类似颜色值压缩为一个记号。

与 PNG 类似，JPG 文件中也包含元数据块，这意味着图像编辑器或者照相机都能在文件中插入一些不需要的数据。[7]

JPG 格式还有一个好处是，大多数移动设备现在有系统可用的 JPG 编码和解码的硬件，这就意味着解码一个 JPG 文件需要的时间要比解码同样大小的 PNG 文件短很多。

12.3.3　GIF

GIF 是另外一种支持透明度的格式，此外它还支持动画（这也是 cats on the internet thing 的直接原因）。GIF 格式文件的生成包含两个压缩步骤，第一步是有损的色彩数量压缩，将整个图像的颜色数量减少到只有 256 种，第二步则是无损的 LZW 压缩。将图像颜色的数量减少到 256 种会极大地降低图像的质量，而好处则是文件压缩得更小，这也会使 LZW 算法的压缩效果更好。网站对 GIF 有很好的支持，然而在本地平台上它没有得到统一的支持。

注 7：这就是我们用手机拍的照片放到社交媒体上能打上地理标签的原因。

12.3.4 WebP

WebP 格式为用户提供了介于 PNG 和 JPG 之间的中间地带。WebP 既支持无损模式和透明度，同时也支持有损模式。总的来说，你可以从 JPG 和 PNG 各自的优点之间进行选择。虽然听起来好像 WebP 就是最好的图像格式，但是它也存在一些问题，最主要的是，浏览器不是 100% 支持它。同时，在开发移动应用程序时，为了使用它还需要包含相应的库（安卓平台除外，因为它本身就支持 WebP）。除此之外，在有损压缩模式下的高压缩率，也就意味着在解压时它要比 JPG 或者 PNG 格式慢一些。

12.3.5 现在，到了选择的时刻

有了上面这些知识，对于给定的图像需要选择哪种格式，我们应该已经有了非常清晰的流程图。

(1) 你需要透明效果吗？

- 如果答案是需要，那么接着问第二个问题：客户端支持 WebP 格式吗？
- 如果支持，则选择 WebP 格式；如果不支持，则选择 PNG 格式。

(2) 如果问题 (1) 的答案是不需要，还是接着问上面的第二个问题：客户端支持 WebP 格式吗？

- 如果支持，则选择 WebP 格式（除非性能方面会出现问题）；如果不支持，则选择 JPG 格式。

互联网上的图像格式争夺战

当谈到互联网上的大量内容时，图像算是目前网络上最大的负载（虽然也有争论说视频才是）。

但真正有意思的是，虽然信息压缩能解决一些信息阻塞问题，但是还有大量人为的问题，让它不能为所有人都带来便利。

我们先回到 1985 年，那一年 Unisys 公司申请了 LZW 压缩算法的专利。

几年后，CompuServe 公司以 LZW 算法为基础开发了 89a 格式（也就是后来的 GIF 格式），当时该公司没有意识到 LZW 算法已申请了专利。Unisys 也没有在意，直到 1993 年 Netscape 浏览器增加了对 IMG HTML 标签的支持，同时也增加了对 89a 格式的支持。一年内，随着动画图像在互联网上流行起来，Unisys 公司开始申请对其专利的保护。最终 CompuServe 公司和 Unisys 公司于 1994 年 12 月在法庭上达成协议，Unisys 公司宣布将对使用 89a 格式的所有软件收取专利使用费。在这一协议达成后的几个月里，一个小组，由 7 名工程师组成，开发了 PNG 这种全新的、无专利权的数据格式。没过几个星期，Netscape 浏览器就全面支持这种新的数据格式。

2004 年，LZW 算法的专利终于到期，但是整整有 10 年，关于 GIF/PNG 这两种格式的争论一直都是网络上讨论的热门话题。[8]

JPG 这种格式成为标准已有一段时间了，同时它也得到了大多数图像编辑程序和浏览器的支持。2013 年，谷歌和其他开源贡献者开发了一种新的图像编解码算法，名为 WebP，其目标是在保持图像质量的情况下，使它压缩得比 JPG 还小。WebP 节省的空间不算大，根据图像大小的不同，与 JPG 相比大约能节省 5%~30% 的空间。然而，这对需要处理大量图像数据的公司来说（比如购物网站和社交网站），由此而节省的空间还是相当可观的。对这些公司来说，节省 30% 的空间就意味着费用的显著下降、传输速度的加快以及加载时间的缩短。

但同时 WebP 格式也面临着一个很大的挑战：让所有的浏览器都支持它。Chrome（由谷歌开发）是第一个支持它的浏览器。然而，更大的挑战则来自于竞争对手 Mozilla 的 FireFox 浏览器，它不仅不支持 WebP 格式而且还公开反对，声称 WebP 不过尔尔，其压缩能力还不如一些 JPG 的变体。事实上，Mozilla 的开发团队甚至开源了一种被称为 MozJPEG 的新编解码算法来改善 JPG 的无损压缩预处理步骤，这一切都是为了阻止 WebP 获得更多的采用。

然而这样的抵制并没有让谷歌停止为 Google+ 以及 Google Play store 实现相应的编解码器。Faceboook 很快也以自己的方式实现了对 WebP 的支持，并总是称赞 WebP 在压缩和图像质量方面的成就。从那之后，WebP 获得了越来越多的支持，甚至成为某些视频游戏压缩的主要部分。

Mozilla 的抵制没有持续太久，到 2015 年，它对 WebP 格式的态度来了个 180 度的大转弯，并公开表示"技术决策往往是个人偏好、政治谋略和公司间竞争的结果，但客观确凿的数据仍然可以赢得胜利。"

这样的表述可以说充满智慧。互联网上图像格式的故事，让我们看到了与压缩有关的技术采用、程序员心理以及客户利益等一些有趣的事实。即使某个算法在技术上是先进的，它还是会受到与此相关的同类技术产品偏见的影响，同时它也必须获得具有普遍怀疑精神的工程师的承认和认同。

12.4 GPU纹理格式

计算机不能直接利用压缩格式的数据绘制图像，而是需要先将压缩的数据加载到内存中，然后再解压为系统可以直接渲染的格式。默认情况下，图像会被解压为每像素 32 位的格式，其中红绿蓝三种颜色通道以及 alpha 通道都是 8 位（也就是 RGBA_8888 这种表现形式）。然后，图像会被当作纹理传输到 GPU 中，也就是说你生成的每一个位图都会同时需要 CPU 和 GPU 内存。结果就是，不论图像在网络上的压缩质量如何，当在设备上显示时，它就会占用大量的内存。

注 8：在这一问题解决后，争论又转移到"GIF"的正确发音上。

好消息是，GPU 能直接渲染的像素压缩格式是存在的，因此你可以利用这一点，将从网络中传输过来的数据解压为这些压缩格式之一，这样 GPU 就可以直接渲染，而无须解压这一步骤。DXT、ETC 和 PVR 就是几种这样的有损像素压缩格式（见图 12-7）[9]。

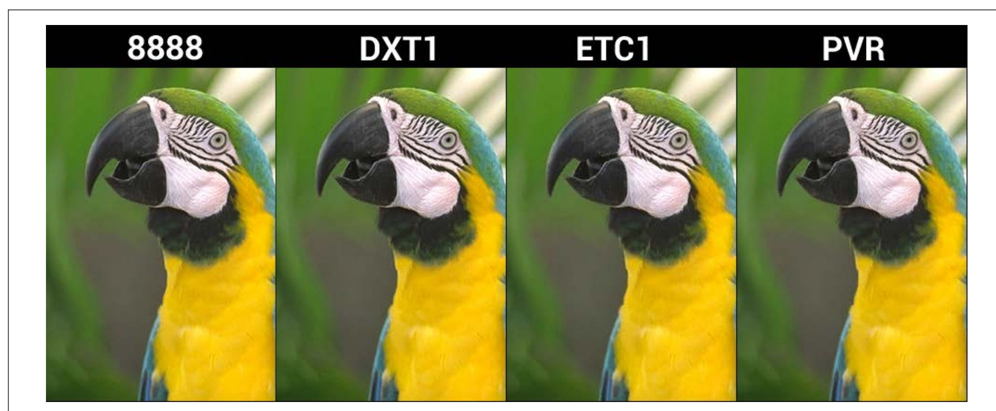


图 12-7：与 RGBA_8888 相比，DXT1、ETC1 和 PVR 这几种有损 GPU 纹理压缩格式所需的空间要更少，同时在很多情况下能保持很高的图像质量

你能想象到，这些 GPU 纹理压缩格式对视频游戏开发人员是多么有用！不仅仅因为视频游戏中包含很多的图像信息，更重要的是几乎可以认为它们需要在 GPU 内存中一直保持可用状态。因此，任何内存占用上的节省对于应用程序的流畅和稳定都很重要。

12.5 矢量格式

通常来说，图像是通过二维网格中的像素来显示的，这些像素表示的是图像本身的颜色。当从远处观察图像时，像素之间的边缘就会消失，这样人的眼睛（被欺骗了）看到的就是平滑的颜色渐变。这种类型的图像通常被称为光栅格式图像（raster format image），它可以（比较直接地）在屏幕上渲染。

但是如果传输的不是最终的图像，而是描述图像是怎样生成的语句呢？这就是矢量图像格式背后的概念。一般来说，这样的格式里包含的是一些程序指令，只要按照顺序执行就会生成最终的输出图像。

图 12-8 显示了同一幅图像的光栅格式表示和矢量格式表示，从中可以看出取舍很明显。

注 9：图 12-7 的彩色图片，请通过本书图灵社区页面（https://www.ituring.com.cn/book/1893）的“随书下载”处获取。——编者注



图 12-8：光栅格式图像（左图）和矢量格式图像（右图）的对比示例。值得注意的是，矢量格式图像更简单，包含的像素细节也更少。这是因为这种图像格式不适合用来生成高质量的数据

矢量格式有一些很有趣的优点。例如，对某些类型的复杂图像来说，比如主要是由线组成的技术图纸，列出其中的点并描述如何连接这些点就要比传输每个像素有效得多。（这也是一种形式的压缩。）此外，矢量图像能精确地放大缩小，当需要在不同的设备上显示同一图像的缩略图、图标和全屏形式时，这一性质相当有用。

当然，付出的代价就是加载的时间。因为需要通过执行指令来为 GPU 生成光栅化的图像，所以，这一格式就是在用客户端的速度换文件的大小。虽然它可以使需要传输的数据变小，但在渲染时由于需要重新生成图像，因而加大客户端的时间开销。

SVG 是一种常用的矢量图像格式。无论源数据多大，有了它，我们就能用很少的内存来描述图像，并在客户端生成高质量的与分辨率无关的图像。当然，SVG 也有一些局限，其中之一是它只能用来表示某种类型的图像质量。也就是说，矢量图像通常比较简单，只会使用一组最基本的类型来定义如何在屏幕上生成颜色。例如，草原上的一片绿地，由于涉及太多复杂的形状，因此如果用矢量图来表示，根本就不会节省空间。

总的来说，矢量格式适用于标志、技术图纸以及简单的图像样式，而光栅格式则适用于相片以及其他信息密集的图像。

下面给出一个示例让你来试试，这一 SVG 文件会生成如图 12-9 所示的图形。

```
<svg height="140" width="140">
  <defs>
    <filter id="f1" x="0" y="0" width="200%" height="200%">
      <feOffset result="offOut" in="SourceGraphic" dx="20" dy="20" />
      <feColorMatrix result ="matrixOut" in ="offOut"
        type ="matrix"
        values ="0.2 0 0 0 0 0 0.2 0 0 0 0 0 0.2 0 0 0 0 0 1 0"/>
```

```
        <feGaussianBlur result="blurOut" in="matrixOut"
          stdDeviation="10" />
        <feBlend in="SourceGraphic" in2="blurOut" mode="normal" />
      </filter>
    </defs>
    <rect width="90" height="90" stroke="green" stroke-width="3"
      fill="yellow" filter="url(#f1)" />
  </svg>
```

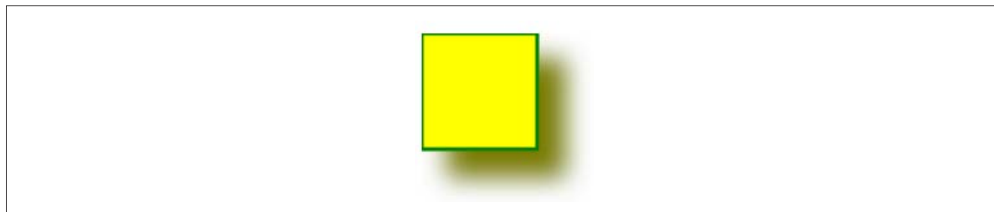


图 12-9：由示例 SVG 文件生成的图形

12.6 收获的捷径

对现代应用程序来说，与用户进行交互的数据绝大多数是图像。运营方经常向用户发送的是缩略图、新的图像、社交媒体推送、朋友的照片以及广告这些内容。用户则会经常上传一些他们日常生活中的照片和内容。当你考虑减少应用程序的数据所占用的空间时，图像应该是你最先关注的。可以说，这是最容易实现的目标，而且常常是小的改变就能带来大的收获。

第13章

序列化数据

除了图像数据，序列化内容是网络应用程序处理第二多的数据格式。压缩图像数据这项工作肯定会事半功倍，但是仔细研究序列化内容也同样重要。

这里所说的“序列化”是什么意思呢？序列化是将高级数据对象转化为二进制字符串的过程（与之相反的过程则称为反序列化）。这一转换可以应用到很多不同的数据类型上，但用它来描述将内存中的结构体或者类转化为能通过网络传输的文件或者内存二进制大对象的过程是最准确的。

这种特定的使用是现代移动和网络应用程序的数据转换中最常见的场景。我们不妨来看一下你最喜欢的社交媒体应用程序。当你第一次加载它时，为了在屏幕上显示正确的信息，在客户端和服务器端之间就有很多序列化数据的传输。当你收到更新、新闻或者信息时，其底层仍然是序列化数据的传输。当你发布自己的最新状态时，这一输入将被写入设备的内存中，序列化后会被上传到服务器端，然后会被反序列化并写入数据库中，接着会再次被序列化以便将更新发送给你所有的朋友。

虽然从文件大小上来看图像是数据压缩的主要对象，但是序列化内容被压缩的次数要更多。

这就意味着，压缩的性能对序列化速度、反序列化速度，以及需要发送给数百万用户的结果文件大小来说至关重要。为了让你有更多的了解，下面来看看两种最常见的序列化文件格式 XML 和 JSON，并讨论一些能让这些文件变得更小的技术。

13.1 了解常见的使用场景

清楚地了解序列化内容是怎样使用的很重要，因为这会对你决定如何压缩数据产生很大影响。我们下面来看看序列化数据最常见的使用场景。

13.1.1 服务器动态生成的数据

这是现代移动应用程序中最常见的序列化数据类型。客户端通常会向服务器发出查询请求，可能请求的是数据库操作的结果，服务器计算出结果，将其内容序列化后再发送给客户端，由客户端进行反序列化。在这个过程中，HTTP 协议栈通常会对序列化数据进行进一步的压缩（例如使用 GZIP），从而使文件变得更小。考虑到节省的空间相当可观，因此客户端付出的解压时间开销是值得的。

13.1.2 服务器拥有的静态数据

虽然动态生成的数据很常见，但是应用程序同样也有静态的序列化内容，例如，向客户端发送最新的配置文件。开发者会在服务器上不定期地更新这些文件，而且通常是离线进行的。这样，服务器就可以认为这些文件是静态的，并在客户端请求时将其发送过去。同样，这些文件也会被 HTTP 协议栈进一步压缩。

13.1.3 客户端动态生成的数据

在很多情况下，客户端也会向服务器发送信息，其中就有客户端本身生成序列化信息的情形。这就意味着序列化操作以及数据压缩过程的所有开销都完全由客户端设备来承担。对便携式计算机或台式计算机来说，这可能不是问题，但对移动电话、平板计算机和可穿戴设备来说，久而久之可能就会造成大麻烦。此外，由于这些设备往往电量有限，因此一般不倾向于将客户端的资源花在要上传的数据的压缩上。这就形成了一种独特的平衡行为，即开发人员需要解决他的应用程序面临的此类问题。

13.1.4 客户端拥有的静态数据

最后，还有一类存储在本地且客户端一直在使用的数据，例如，布局信息就是一次编写然后被多次加载，而且也没有什么变化。这样的信息很容易压缩，而且通常是在应用程序的构造期间，此时有额外的电源可用。客户端唯一需要做的，就是保持数据驻留（或者永久地存储起来）并在需要时将其加载到内存中。

13.2 序列化格式的问题

目前，两种最常见的序列化格式就是 JSON 和 XML，这主要是由于在过去的 20 年中它们

被网络平台广泛地使用。虽然易用并且很流行，但是这两种格式还是遇到了一些非常特殊的压缩问题。

13.2.1 可读文本

JSON 和 XML 这两种格式最吸引人的地方在于，它们（或多或少）是可读的。也就是说，如果用文本编辑器打开序列化后的文件，就可以读取全部内容，如下随机选取的 JSON 代码片段所示。

```
{
  "base": {
    "reboot": { ...omitted for brevity... },
    "updateBaseConfiguration": { ...omitted for brevity... }
  },
  "robot": {
    "jump": {
      "parameters": {
        "height": {
          "type": "integer",
          "minimum": 0,
          "maximum": 100
        },
        "_jumpType": {
          "type": "string",
          "enum": [ "_withAirFlip", "_withSpin", "_withKick" ]
        }
      }
    },
    "speak": {
      "parameters": {
        "phrase": {
          "type": "string",
          "enum": [ "beamMeUpScotty", "iDontDigOnSwine", "iPityDaFool",
                    "dangerWillRobinson" ]
        },
        "volume": {
          "type": "integer",
          "minimum": 0,
          "maximum": 10
        }
      }
    }
  }
}
```

正如你看到的，它通过将文件表示为一组字符串值，并用标记拼凑在一起，来定义事物是如何关联的。

其优点在于格式十分灵活（绝大多数数据结构可以找到序列化为这种格式的方法），但缺点也很明显，为了保证可读性，其中包含了大量的冗余信息。

只需看一看前面的 JSON 代码片段，我们就会发现为了使整个文件可读，其中包含了大量的空格、换行符和双引号。因此，需要编码的文件比实际需要的要大。当有数值时，这个问题更严重。例如，如果序列化的文件中含有“3.141 592 653 589 793”这个值，那它会至少需要 17 个字节（由于编码格式的不同，因此占用的字节数可能会更多）。考虑到用浮点数来表示这个值只需要 8 个字节（或者 64 个二进制位），这是很大的浪费：人类可读的表示方法需要的空间比二进制方法的两倍还多。

13.2.2　解码时间长

需要注意的是，这样的文本格式在解码时常常会需要很长时间。其原因是多方面的：一是字符串的输入必须经过强力操作才能转化为内存对象（例如将 ASCII 符号转换为整数就不那么容易）；二是在加载期间将数据保存在临时内存里并非总是高效的；三是对旧格式的兼容也会使得编码和解码变慢。

由此带来的另外的问题是，像 XML 和 JSON 这样的格式都倾向于有较长的加载时间，以便客户端能正确地反序列化。事实上，存在不少 XML 和 JSON 编码器专注于减少特定组织的文件类型的加载时间。

13.3　更小的序列化数据

有了这些知识，我们就可以有一些技巧减小发送给用户的 XML 和 JSON 文件的大小。

13.3.1　使用二进制序列化格式

最简单有效的方法就是将 XML 和 JSON 格式扔到一边，找出一种二进制序列化格式来代替它们。虽然二进制格式不再具备 XML 和 JSON 的可读性，但是能保证数据用紧凑和高效的二进制形式编码。这样，文件就会变小，加载速度也会变快。

虽然二进制序列化的格式很丰富，但是我们最喜欢的一些格式处于文件大小和解压时间平衡的位置。如果你想定义自己的格式，那么 Protobufs、Flatbuffers 和 Cap’n Proto 应该是你最先评估的几种格式。

但是如果你还没有打算放弃 XML 和 JSON，或者你的老板不让你放弃基于文本的序列化格式呢？也有很多方法可以让 JSON 数据序列化时效率更高，同时也能让它变得更可压缩。像 BSON 和 MSGPACK 这些格式虽然保留了 JSON 的模式，但在编码时能提供二进制的大小。这可以让你得到更小的文件，而无须改动大量的代码。

这些二进制格式的真正优点在于，与人类可读格式相比，它们可以产生更好的压缩效果，并且在某些情况下，实际上还可以使用通用编码工具如 GZIP 对它进行进一步压缩。

13.3.2 重构列表以获得更好的压缩

下面这点很有意思：当你对数据进行序列化时，其实大多数时间，你所做的只是将数据内容映射为内存对象形式。我们来看看下面这段代码，想想左边的结构体是怎样序列化为右边的 JSON 代码的。

```
struct {                {
    int id;                 "id": 25,
    char* name;             "name": "Hooty McOwlface",
    int gender;             "gender": 27,
    int age;                "age": 88,
    char* address;          "address": "1600 Amphitheatre Pkwy, Mountain View, CA 94043"
    int employeeID;         "employeeID": 3025
}                       },
```

JSON 文件中属性的排序往往与内存中相应结构的表示相同，虽然这对程序员来说易于维护，但是由此生成的整个结构的列表不能产生最好的压缩结果。

首先，由于 JSON 对象（可以随便选一个观察一分钟）是由键值对组成的，由于该结构体的每个实例中都包含相同的属性，因此对整个文件来说包含大量的冗余信息。在下面这个包含人名及国籍的列表中，对每个人都必须重复“name”和“country”这两个关键字。

```
...
{
   "name": "Joanna",
   "country": "USA"
}{
   "name": "Alex",
   "country": "AUS"
},
{
   "name": "Colt",
   "country": "USA"
}
...
```

对以这种形式列出很多元素的大的 JSON 文件来说，“name”和“country”这两个关键字的重复出现使最终的文件变大很多。

其次，回想一下，像 GZIP 这样的编码工具在最初的转换步骤中使用的是 LZ 算法，这就意味着如果能在搜索窗口中发现重复的数据模式的话，LZ 算法就能充分发挥作用。

想象一下，在一个文件中全是这样的雇员数据，并且潜在的重复值之间还存在着空白。例如，在序列化的文件中，一个“age”值离下一个“age”值可能有一两千个搜索窗口。

只需要简单地对列表内容重新排序，你就能解决属性的重复和相似属性值之间的距离这两

个问题。如下面的示例所示，你可以将前面的数组结构[1]（array structure）转换为某个给定属性的所有值都包含在一个数组中，并紧密地放在一起。[2]

```
{
    "name": ["Joanna", "Alex", "Colt"],
    "country": ["USA", "AUS", "USA"]
}
```

这既减少了冗余，同时也使 LZ 算法更容易找到匹配。

用编程的语言来就是，对于大的序列化文件，将结构的数组转换为数组的结构极为重要。因此，当你需要处理大的 JSON 或者 XML 文件时，建议你认真考虑这样的转换。

13.3.3 组织数据以便高效获取

还可以进一步扩展结构转换这一概念。真的需要从服务器中获取完全结构化的数据吗？还是可以分别请求每种类型的数据（如果有必要的话，再在客户端将这些数据汇总起来）？

当前，后端应用程序往往喜欢为所有的用户都提供通用的 API。虽然这对后端系统来说是合理的策略，但对客户端来说并不那么友好，因为这样做的最终结果是，应用程序将传输和处理数据的任务放在了客户端的小型设备上，而实际上有些计算在服务器上处理会更高效。

如果你的应用程序显示的是混合内容，那么你需要确保客户端通过一次请求就能获取所有信息，同时保证服务器返回的数据适合分段缓存。一般来说，你希望客户端能识别出实体，以便将其持久地存储起来，同时避免同样的对象在内存中出现。

在获取这种类型的数据时，很多 API 返回的是非正规化的分层数据。虽然这是大多数网络客户端的首选方法，但对移动客户端来说不太适用，因为移动客户端需要的是持久的数据以及能从本地存储中获取服务。

与其返回分层的数据，不如返回规范化的数据。

我们来看看下面这个不好的例子，同样的 `user_id` 和 `user_name` 在很多地方重复出现。客户端获取到这样的数据后，需要先将大的对象分解，提取出内部的用户对象并去掉重复的，然后再将剩下的对象存储到本地的数据库或内存缓存中。

```
{
"messages" : [{
    "from" : {
```

注 1：从技术上来说，准确的英文表述应该是“array of structs”，或者称为“数据对象的列表”（a list of data objects）。

注 2：值得指出的是，这并不是序列化内容独有的概念，你如果以前处理过与 CPU L2 缓存驻留有关的运行性能问题，就知道那里用的是同样的解决方法。

```
                "user_id" : 1,
                "user_name" : "claude",
                ....
             },
             "text" : "hello
hello",
             "date" : "123"
 },
 {
     "from" : {
     "user_id" : 1,
     "user_name" : "claude",
     ....
   },
   "text" : "how are you",
   "date" : "124"
   },
   {
      "from" : {
      "user_id" : 1,
      "user_name" : "claude",
      ....
   },
   "text" : "you there",
   "date" : "125"
   },
   {
        "from" : {
        "user_id" : 1,
        "user_name" : "claude",
        ....
    },
    "text" : "hello
hello",
    "date" : "126"
    }]
}
```

我们再看一个好的例子。

```
{
"users" : {
        "1" : {
     "user_id" : 1,
     "user_name" : "claude",
     ....
        }
},
"messages" : [{
    "from" : 1,
        "text" : "hello
hello",
        "date" : "123"
 },
```

```
 {
     "from" : 1,
  "text" : "how are you",
  "date" : "124"
  },
  {
      "from" : 1,
   "text" : "you there",
   "date" : "125"
   },
   {
       "from" : 1,
    "text" : "hello
hello",
    "date" : "126"
    }]
}
```

这对客户端来说要容易得多，因为每个对象都只出现一次。同时服务器返回的“用户”信息很容易就能提取出来，用来更新数据库和内存缓存。

不过还可以做得更好，使数据完全不再分层，不信可以看看下面的方法。信息还是那些信息，不过不再有重复，而且更紧凑、更容易直接处理。

```
        {
"users" : {
        "1" : {
     "user_id" : 1,
     "user_name" : "claude",
     ....
        }
},
"messages" : {
"from": [1,1,1,1],
"text": [ "hello
hello","how are you","you there","hello
hello"],
}
```

客户端对需要显示的数据掌握的信息越多，效率就越高。应用程序决定缓存或者删除哪些数据，例如，怎样在新数据达到时使布局无效。移动客户端要比简单的 HTML 渲染器更强大，你完全可以将比较好的结构化数据交给它来处理。

13.3.4　将数据切分为适当的压缩格式

一般来说，像 JSON 和 XML 这样的序列化格式，其实是多种类型数据的“杂物抽屉”。你可以将整数、字符串、浮点数，甚至是图像和声音数据都放进去，然后全部编码为这种效率不高的序列化格式。

然而，将这些大的数据类型单独分开然后再压缩，产生的压缩效果肯定要比将它们放在文件中后再压缩好。不妨认真想一想。如果你需要压缩一个包含 2600 个倒排索引的 JSON 文件，那么 GZIP 肯定帮不了太大忙。不过，如果先将索引分开进行增量编码，就可以有显著的改善。

这同样适用于图像。有段时间出现了一种很可怕的趋势，在设计响应式网页时将 base64 编码的 PNG 文件（即用 ASCII 字符串格式来表示二进制数据）放在 CSS 文件中。这样做的理由是，传输包含了图像信息的 CSS 文件与传输不包含图像信息的 CSS 文件之间的时间差，要比使用网络额外传输缩略图所花的时间要短。除了极少数情况外，我们不能容忍移动应用程序中出现这样的操作。

当你想尽办法想为用户创造快乐时，数据压缩可能不会第一时间出现在你的脑海中。我们想强调的是，它应该出现，至少是在每天的某些时刻你能想到。就像应用程序的其他基础架构一样，如果你能将数据压缩融入到应用程序的开发过程中，结果肯定是事半功倍：它不仅可以让用户更高兴，可能还会带来更高的盈利。

无论你是准备用内置的压缩工具，还是根本不用压缩工具，或者不同的数据类型用不同的管道来压缩，重要的是根据你所掌握的数据有意识地做出选择。

将图像压缩和数据序列化的工作做好，有助于应用程序安全度过其生命周期。如果在开发工作的初期就对数据压缩有正确的认识，有助于始终保持客户端的“轻盈”状态。因此，要从一开始就做起，而不是在最后……你说对吧？

第14章

有损数据压缩

或许，你已经注意到本书花了**大量**篇幅讨论**无损**压缩算法，也就是说解码后的数据与源数据的每一位都相同。

然而，在应用程序的日常运行中，大多数你真正担心的内容其实是用**有损**压缩工具压缩的。在像图像、声音以及视频这些内容中包含的信息超出了人类的视觉系统和听觉系统能（或者需要）处理的范围，即使通过有损压缩格式去掉了额外的信息，人们依然能享受到良好的视觉和听觉体验。

有损压缩工具通常会被首先应用，以减少数据的动态变化范围，从而为进一步的无损压缩做准备。

必须要清楚：有损压缩工具其实有无限多种，选择哪一种取决于需要处理的数据类型、你的需求以及用户愿意容忍多大范围的失真。实际上，它才是数据压缩领域内最富饶的土地，因而能做的事情还有很多。

那么，为什么没有在本书中更多地讨论有损压缩算法呢？

其实理由也很简单……因为那会是另外一本书。

第 15 章

让世界变得更小

15.1　数据压缩与你

不知不觉中你已读到了本书的末尾，前面的章节详细地介绍了那些源于 20 世纪 60 年代的算法，这些算法对当前的计算与技术仍然有着显著的影响。然而，情况又会如何发展呢？很多工程师会高兴地举双手赞成，认为压缩是一个已解决的问题，或者对自己来说压缩并非一项特别重要的能力。不过事实是，在未来的几十年里，数据压缩的重要性丝毫不会比以前差。因此，研究一下数据压缩是如何与你、你的公司以及未来的技术相互关联的，是值得的。

15.2　数据压缩与盈利

当涉及你以及你的公司时，通常是钱的问题。数据压缩与公司的盈利是如此紧密地交织在一起，以至于拥有这方面技术的公司能节省如下费用：

- 用户获取与保持
- 运行成本
- 提前规划

下面来更详细地讨论。

15.2.1　用户获取与保持

网页的加载速度与用户转化率之间有着直接的关系。如果网页的加载速度不够快，用户就会放弃所有在做的事情，包括购买你的产品。反过来说，如果你将产品页的数据压得越

小，网页加载的速度就会越快，那么用户购买以及下次继续访问的可能性就会增大。

统计数据显示，如果移动页面加载超过 4 秒钟，那么平均每 4 个用户中就会有一个用户放弃浏览该页面。这样的测试也值得你的网页去做，因为这意味着仅仅从网页加载方面就能对公司的盈利产生巨大的影响。如果你需要更具说服力的证据，可以看看下面这些真实的故事。

- 亚马逊的数据显示，网页的加载速度每慢 100 毫秒，其收入就会下降 1%。或者，换一个角度来看，对像亚马逊这样的商业巨头来说，如果网页加载速度慢了 1 秒，其收入就会减少 16 亿美元。反过来看就是，亚马逊的网页加载速度每快 100 毫秒，其收入就会增加 1%。
- 沃尔玛的最新报告显示，其页面加载速度每加快 1 秒，它的客户转化率就增加 2%。网页加载速度每提升 100 毫秒，公司的收入就会增加 1%。
- Shopzilla 网站将其网页平均加载速度由 6 秒缩短为 1.2 秒后，其收入增加了 12%，同时网页浏览量增加了 25%。
- 像 AutoAnything 这样的小网站将加载时间缩短为原来的一半后，其收入增加了 13%。

此外，更好的网站评论、更多的下载口碑和更好的客户保持，可以为营销部门节省大量的资金。

15.2.2 运行成本

网站上的大量内容必须存储在某个地方，通常来说其实就是云服务器上大量的硬盘。硬盘要花钱，向云端传输和下载数据也要花钱，租用（或建立并运营）数据中心和带宽还是要花钱。即使云技术已经标准化且其费用在大幅下降，带宽和存储仍然是大公司面临的重要财务挑战。

2015 年，Netflix 宣布开始调整其视频流压缩技术，要根据内容本身的噪声来决定视频使用哪种压缩算法。此举旨在节省公司的大量带宽成本，同时最大限度地利用各种特定设备的性能。

同样在 2015 年，Facebook 公布了其提供图像预览服务的细节，每幅图像的预览只有 200 字节。考虑到社交媒体网络每天传输的图像的数量，这对他们来说是一个很大的成功，特别是对使用 2G 设备的用户来说。

要理解这一点不需要你学过高等数学：数据量更小，就意味着更小的出站流量成本、更小的入站流量成本以及更小的存储成本。

15.2.3 提前规划

网页正在变得越来越大。自 2011 年以来，HTTPArchive 这个独立网站就一直在对排名前 1000 位的网站进行下载、编目、汇总统计的工作。在分析过程中得到的一个很有趣的统计数据是网页的平均大小。该统计对显示页面信息需要的字节数进行了汇总，其中包括 JavaScript、HTML、CSS、JPG 和视频文件的大小。统计显示，页面平均大小在 2013 年增长了 24%，达到了 1.5 MB，而到 2015 年则变成了 2 MB。

网页之所以一直在变大，其中一个原因是它们包含的图像越来越多，同时每幅图像也变得越来越大。此外，由于网页变得越来越复杂，因此网页中包含的代码也越来越多。对 2G 用户来说，这一问题正变得越来越突出。为了解决这一问题，谷歌这样的大公司推出了移动页面加速（Accelerated Mobile Pages，AMP）这一全新的框架，专门用来减小站点加载时依赖的图形、图像文件的大小，在带宽有限的情况下为用户提供更精简、加载速度更快的内容。

15.3 让用户的生活更美好更便宜

移动设备已成为现代生活中很重要的一部分。想到用户的体验是如此严重地依赖于商品目录的图像是以多快的速度从服务器端传输到用户端，真让人感慨。

当我们试图降低出站流量的成本时，用户也在想着降低入站流量的成本。我们必须清楚：一切都是需要用户支付费用的。他们中的大多数人要为数据付费，以兆字节为单位，并且付费惊人。

mobiForge 在 2013 年进行了一项小型的数据分析，显示了使用高速数据套餐的成本。当时，AT&T 的漫游资费为 12 美元 /MB，浏览 microsoft.com 的用户平均每次需要支付 17.5 美元给 AT&T。

我们不仅要考虑金钱成本，还要考虑电池的开销。连接速度较慢的用户在下载同样的内容时需要的时间也较长，这就意味着使用电池的时间也要比连接速度快的用户长。结果就是，连接较慢时，用户消耗电池电量的速度变快了。

数据压缩与这两个问题都直接相关。在连接速度同样慢的情况下，文件越小就意味着下载需要的时间会越短，消耗的电量也会越少。最终的结果就是，用户能快速看到商品的图像。

15.4 对下一步技术的思考

如果你很幸运，能生活在一个网络连接状态很好的国家，那么恭喜你！很有可能你在为一个相当不错的移动连接网络支付着很合理的费用。然而，从世界范围来看，情况并非如此，更多的人在使用着连接状况糟糕的网络。展望未来，移动计算的明天将由首次使用移动网络的 50 亿人以及他们使用的移动网络的质量决定。

15.4.1 未来的50亿用户

现在全球大约有 74 亿人，其中大约有 20 亿人是互联网用户。其余的大部分人则生活在互联网连接快速发展的国家。这意味着，未来你的用户的最大来源是亚洲和非洲的新兴市场。过去 10 年移动计算技术的快速发展告诉我们，下一个 10 亿用户将第一次主要是通过移动电话，而不是台式计算机或便携式计算机接入网络。

Eric Schmidt 和 Jared Cohen 在 *The New Digital Age* 一书中很好地阐述了这一主题。

> 在非洲已有超过 6.5 亿的手机用户，而在亚洲手机用户的数量已接近 30 亿。他们中大多数人使用的是只有基本功能——电话和短信——的手机，这主要因为他们所在国家的数据服务费用十分昂贵，即使是那些买得起具有网络功能的手机或智能手机的人，也负担不起数据服务的费用。这种状况将来肯定会改变，一旦这种改变发生，这些人就会深深受益于智能手机革命。

15.4.2 移动网络

在这些具有高潜力的国家建立网络并不便宜。考虑到 Verizon 只是将其网络升级到 4G 就要花费 500 亿美元，因此为这么庞大的人口建立一个全新的网络，其成本肯定会是天文数字。同时由于牵涉政府，这一费用通常还会膨胀（对政府和电信公司来说，这样的情况很常见），而这又常常会导致所有的成本转嫁给终端用户。

在过去的几年里，全世界的网络速度都有了很大的提高。然而，同时还要看到这种改进在数量上或地理位置上不平衡。Google Analytics 有一组奇妙的图表展示了世界上网络连接的趋势。从中可以很容易地看出，改进的想法并非一成不变。

简而言之，移动网络的速度仍然会继续提高，尽管提高的速度会比较慢，在各地也不均衡，而且花费巨大。如果你期待移动网络突然变得更快，很可能就需要找一把舒适的椅子慢慢坐着等待。

例如，2G 网络的传输速度大约为 0.021 MB/s，而 GZIP 的压缩速度则可达 61 MB/s，即使 GZIP 的压缩速度降为原来的十分之一，压缩 1 MB 的速度仍然比通过网络传输要快。本书作者柯尔特对这些数据的分析表明，**与投入数百万美元升级网络硬件相比，投资更好的压缩解压编解码器要划算得多。**

15.5 开始行动

到这里，画面应该已经相当清晰了。下一次大规模的计算革命很有可能发生在人口众多的地区，这些地区的人不是很富裕，这就意味着他们在选择移动硬件和手机供应商时倾向于选择较慢的硬件和网络。

但他们同样也有对快速传输数据的需求，而且开发人员之间同样会为拉拢这些用户而竞争。移动计算的趋势仍会继续，而移动应用程序的平均数据成本也会继续飙升。虽然他们玩的仍然是追赶游戏，但他们已然远远落后。发送 25 个缩略图或者加载满页有图片的新闻对他们来说成本太高，加载也太慢，这会导致他们放弃缓慢的体验而去追求更快的体验。

2015 年这一问题是如此重要，以至于像 Facebook 这样大的开发公司推出了精简的、2G 友好的版本，以减少在这一新兴移动市场中获取用户的障碍。

作为开发人员，你既不能真正控制网络，也不能控制硬件。你唯一能控制的只有数据，你可以做大量工作，以确保数据被压缩得很小，这样它们就能以较高的质量、较快的速度传输给用户，从而让用户获得正常的计算体验，并且一直是你的应用程序的忠实用户。

那还等什么呢？

数据压缩术语表

7-Zip

一种拥有极高压缩比的压缩软件。

alpha 混合（alpha blending）

将图像与背景叠加从而产生部分或者完全透明效果的过程。

alpha 通道（alpha channel）

除红、绿、蓝这 3 种颜色通道之外的附加通道，用来表示像素的透明度，其取值范围为 0~1。

alpha 透明度（alpha transparency）

alpha 通道中传输的像素透明度值。

BMP

一种位图文件格式，位图文件是有红、绿、蓝这 3 种颜色通道的简单光栅编码图像文件。

BSON

JSON 序列化格式的二进制版本。

BZIP/BZIP2

使用伯罗斯 – 惠勒变换压缩文件的免费开源文件压缩程序。

Cap’n Proto

一种二进制序列化格式。

DEFLATE

一种利用 **LZ 算法**和**统计编码**来实现压缩的流行算法。

Flatbuffers

一个高效的开源跨平台序列化库，由谷歌开发，用于游戏及其他性能关键的应用程序。

GIF

一种图像压缩格式，以支持 alpha 透明度且广泛用作动态图像而闻名。

GZIP（**GNU zip**）

互联网上广泛使用的不受专利保护的压缩程序。

H.264

一种视频编码格式，是目前最常用的录制、压缩和分发视频内容的格式之一。H.264 因其是蓝光光盘和视频流的编码标准之一而广为人知，该格式已申请专利保护。从严格的数学意义来说，H.264 通常用于有损数据压缩，尽管损失量有时可能让人难以察觉。当然，H.264 格式也可以用来创建真正的无损数据压缩，例如，在有损编码的图片中设置局部的无损编码区域，或者是支持完全无损编码这种罕见情况。

HTTP 协议栈（HTTP protocol stack）

组成超文本传输协议（HyperText Transport Protocol，HTTP）的一组协议。

ITU

国际电信联盟（International Telecommunication Union）的简称。

JPG/JPEG

广泛应用于数字图像领域的有损数据压缩格式。

JSON

JavaScript 对象表示法（JSON）是一种数据交换格式，除了便于计算机解析和生成之外，同样便于人类读写。

LZ77、LZ78

通过引用未压缩数据流中先前存在的数据的单个副本来替换重复出现的数据来实现压缩的一系列算法。其他的实现包括 LZFSE、LZHAM 和 LZTurbo。

LZA

基于 **LZ77** 算法的文档压缩工具。

LZMA

基于 **LZ77** 算法的文档压缩工具。

LZ 与 LZW 算法（Lempel-Ziv and Lempel-Ziv-Welch algorithms）

一系列从数据流找出分词（tokenizing data streams）的无损算法。

详见第 7 章。

MSGPACK

一种小而快的二进制序列化格式。

Mumbo jumbo

斐波那契编码里的内容。

***n* 元语法**（*n*-grams）

给定文本或语音中连续出现的 *n* 个语词。

Protocol buffers, protobuffs

谷歌公司开发的与语言和平台无关的、可扩展的、用于序列化结构数据的格式。

Rate

每个符号的平均熵。

tANS

Jarek Duda 在论文 *Asymmetric Numeral Systems: Entropy Coding Combining Speed of Huffman Coding with Compression Rate of Arithmetic Coding* 中描述的一种不对称数字系统或算术数字系统（arithmetic numerical systems）的变体。

详见第 5 章。

XML
XML 代表可扩展标记语言，通过定义一组规则将文档编码为人和机器都可读的格式。

XOR
异或，参见词条**按位异或**。

ZIP
一种支持无损数据压缩的文件格式，它不是压缩算法。

按位异或运算（bitwise exclusive OR，XOR）
其中的按位（bitwise）是指对每位的操作都是独立的；异或（exclusive OR）则是一种逻辑运算：当且仅当两个输入相异，即其中一个为真另外一个为假时，输出才为真。

编解码器（codec）
Codec 是 coder-decoder 的缩写，是能够对数字数据流或信号进行编码或解码的设备或计算机程序。

编码理论（coding theory）
研究编码方法的科学，以提高在噪声信道上进行数据通信的效率，并降低错误率，从而最大限度地利用信道以接近信道容量。编码可大致分为两类：数据压缩（信源编码，source coding）与纠错技术（信道编码，channel coding）。加密算法则是信息论中的第三类编码。

编码器（coder）
见词条编码器（encoder）。

编码器（encoder）
压缩软件的一部分，负责将源信息转换为压缩数据形式。

变长编码（variable-length codes，VLC）
使用不同长度码字的编码方法，一般来说，最短的码字被分配给数据集中最常见的符号。

详见第 4 章。

伯罗斯 – 惠勒变换（Burrows-Wheeler transform，BWT）
也称为分块排序压缩（block-sorting compression），它是一种可逆变换，其做法是将字符串重新排列，使相同的字符聚在一起。

详见第 8 章。

部分匹配预测算法（Prediction by Partial Matching，PPM）
一种基于马尔可夫链的算法，有多种变体，包括 PPM*、PPMD 和 PPMZ。

详见第 9 章。

程序综合（program synthesis）
自动生成满足一组特定要求的程序。

词典顺序（lexicographic order）
在词典中，词条的顺序是根据其组成字母的字母顺序排列的。

错误纠正（error correction）
将代码附加到信息内容后，以便检测和纠正数据传输中的错误。错误检测和纠错码提高了信息的可靠性，使其更能够适应噪声传输环境。错误纠正与数据压缩是正交的。

单词查找树（trie）
在计算机科学中，单词查找树又被称为“数字树”（digital tree），有时也被称为“基数树”（radix tree）或

“前缀树”（prefix tree，因为可以通过前缀去查找），是一种用于存储动态集或关联数组（其键通常是字符串）的有序树结构。

定量化（quantification）
把人类的感官观察和经验映射成一组数值的计数和测量行为。例如，用分贝表示音乐会上的噪声音量。

多上下文编码算法（multicontext coders）
将多个符号和统计表或者模型结合在一起的算法，以便确定编码下一个符号所需的最小二进制位数。

详见第 9 章。

多重集合（multiset）
是指同一元素多次出现的集合。

二分查找（binary search）
一种在有序的数组中查找目标值的算法。

二进制计数系统（binary or base 2 number system）
一种仅使用数字 0 和 1 表示数值的方式，数值中的每一位都是 2 的幂，例如十进制数值 5 的二进制表示为 101，这是因为 $2^0 + 2^2 = 5$。

二进制删除信道（binary erasure channel）
一种用于**信息论**分析的通信信道模型，其背后的思想是一个二进制位永远不会错，它要么存在且正确，要么就是被“删除”了。

非对称数字系统（asymmetric numeral systems，ANS）
统计压缩的一种现代变体，在早期已显示出接近熵压缩的前景，其性能则可与**哈夫曼编码**相媲美。

详第 5 章。

非奇异码（nonsingular codes）
指在信源编码中，所有信源符号都映射到不同的非空位序列（码字）。

分块（blocking）
为了更好地压缩而将一组数据分为更小的“块”的行为。

分块排序压缩（block sorting compression）
能够高效完成伯罗斯 – 惠勒变换（Burrows-Wheeler transform）的应用程序名称，该程序先将数据流分块，然后对每块数据而非对整个数据流进行伯罗斯 – 惠勒变换。

分组（grouping）
在数据压缩领域，指的是将二进制位数分配给一组符号而不是单独的符号。例如，在处理文本时，我们可以对 100 个最常见的单词进行编码。找出要分组的字符串或子字符串本身就是一个挑战。

分组码（block codes）
任何将数据分块编码且具有纠错功能的编码方法。

概率分布（probability distribution）
一种统计函数，用于描述随机变量所有可能的取值范围以及取值的可能性。该取值范围会介于统计上可能的最大值与最小值之间，但出现在概率分布图上的可能值则取决于众多因素，包括分布平均值、标准偏差和**偏态**（偏态请参阅 Investopedia 网站上的文章 *Skewness*）。

归一化（normalization）
将不同数值范围的值调整为概念上相同的范围。

哈夫曼编码（Huffman coding）
一种基于前缀的无损数据压缩编码。

详见第 5 章。

行程编码（run-length encoding，RLE）
RLE 充分利用来数据流中存在的相同符号连续出现的现象，并使用由符号值与重复次数组成的元组来代替连续相同符号构成的“行程”（run）实现压缩。

详见第 8 章。

互信息（mutual information）
是对两个随机变量之间共有信息的度量。

加密算法（cryptographic algorithms）
为了使内容保密、传输更安全而对信息进行编码的算法。

键值对（key-value pairs）
将数据表示为成对的集合，例如，[词语 , 定义] 或者 [行 , 值]。

解码器（decoder）
压缩软件的一部分，负责将压缩后的数据流转换为未压缩的数据。

纠错码（error-correcting code）
附加到信息内容后的代码，可以简单到只是确认信息内容是正确的，也可以复杂到足以“修复”其中的错误。

局部偏态（locality-dependent skewing）
这个词是我们自造的，想表达的意思是，在数据流的不同位置某个字符出现的频率很不相同。

卷积码（convolutional code）
一种用于提高数据传输可靠性的纠错码。

克劳德 · 香农（Claude Shannon）
美国数学家，被认为是“信息论之父”，也是这本书能够存在的原因。因此，不妨读一读维基百科上介绍他的文章，看看他怎样让我们的生活变得一团糟。

拉普拉斯估计（Laplace estimator）
一种计算公式，用于估算观察值很少时或在（有限）样本数据中根本未观察到的事件的概率。

量化（quantization）
将连续值（如实数）强制转换为相对较小的离散值（如整数）的过程。

量子位（Qbit 或 qubit）
Qbit 或 qubit 是 quantum bit 的缩写，量子位是量子计算机中信息的基本单位，也用来表示量极少的东西。

列表译码（list decoding）
其背后的主要思想是，解码算法并非只输出单个可能的消息，而是输出可能的消息列表，其中包含正确的消息。与唯一可译码相比，这样就能够处理更多的错误。

逻辑综合（logic synthesis）
在电子学中，将所需要电路行为的抽象形式转换为用逻辑门来设计实现的过程。

马尔可夫链（Markov chain）
以俄罗斯数学家安德烈 · 马尔可夫命名的一种随机过程，可以从状态空间中的一个状态转移为另一个状态。该过程必须满足如下性质：下一个状态的概率分布仅取决于当前状态，而与之前的状态序列无关，这一性

质通常被称为“无记忆性”。

排列（permutation）
在数学中，排列是将集合中的所有元素重新排成某种序列或顺序的行为。

前移编码（move-to-front encoding，MTF）
一种局部自适应的数据转换算法。

详见第 8 章。

前缀编码（prefix code）
如果字符集中一个符号对应的码字不是其他任何符号对应码字的前缀，则称该编码为前缀编码。这就意味着在接收到完整码字后可以立即解码，前缀编码始终是**非奇异的**且是**唯一可译的**。

前缀性质（prefix property）
这一性质规定在将某个码字分配给一个符号后，任何其他码字都不能够以该码字开头。**变长编码**必须具备这一性质。

区间编码（range coding）
一种与算术编码基本相同但没有专利限制的算法。

冗余信息（redundancy）
去掉后不影响意义或者功能的词语或数据，也就是重复或多余的信息。例如在句子“池塘里有十（10）只鸭”里，“（10）”就是冗余的。

在信息论中，冗余等于传输消息所需的二进制位数减去消息中实际信息的二进制位数。

熵（entropy）
参见词条**信息熵**。
上下文模型（context modeling）
利用与一段数据相关的多个信息信号来推断最适合应用于其上的压缩算法类型的过程。

上下文压缩方法（contextual compressor）
根据当前上下文中输入符号的概率来确定输出符号的压缩方法。

详见第 8 章。

视频编解码器（video codec）
压缩或解压缩数字视频的电子电路设备或软件，能够将原始的未压缩数字视频转换为压缩格式，也能够将压缩后的数据转换为原始的数字视频。在视频压缩中，“编解码器”（codec）是“编码器”（encoder）与“解码器”（decoder）的联合，只能压缩的设备通常称为编码器，而只能解压缩的设备则称为解码器。

数据集的变化范围（dynamic range of data set）
本书中指的是表示数据集中的每个值所需要的二进制位数范围。

数据流（data stream）
只能够在某个时间点以小规模形式获取，而不能一次性访问每个部分的数据块。现实生活中的示例是在收音机前听音乐。

数据流分词（tokenizing a stream）
为数据流的内容分配符号。例如，在词法分析中，分词（tokenization）是将文本流分解为单词、短语、符号或其他有意义的元素，这些元素都称为词条（tokens）。而在数据压缩中，分词指的是找到为数据流生成理想“单词”字典的最佳方式。

数据压缩（data compression）
用比原数据流更少的二进制位数来表示其中信息的方法，例如将“Rolling on the floor laughing”（笑得在地上打滚）缩写为“ROFL”时，就将原来的 29 个字符压缩为 4 个字符，节省了 86% 的存储空间。（这也暗示了知道上下文可以提高压缩率的事实。）

数据转换（transformation of data）
在数据压缩中，数据转换指的是在不改变信息内容的情况下改变数据流，使其更容易压缩。例如，对数据流 [123456,123457,123458] 而言，从 N 到 N+1 的增量需要的位数就要比 N+1 少，利用增量表示法可将 [123456,123457,123458] 表示为 [123456,1,1]。不过为给定的数据流找到正确的转换本身就是巨大的挑战。

算术编码（arithmetic coding）
通常，编码算法用较少的位存储数据中的常用字符，而用较多的位存储不常用的字符，从而使总的存储位较少。与通常的编码算法不同，算术编码不按 1:1 的比例为每个字符分配一个码字进行编码，而是将整个输入流从一组符号转换为一个长度通常很长的数值，其则表示与整个输入流真正的熵值很接近的真实值。

详见第 5 章。

通道（channel）
信息传输的途径。

通信（communication）
两个或两个以上的参与者之间进行的有目的的信息交换活动，要传达或接受的信息则是通过由符号与符号规则组成的共享系统来传递的。通信的基本问题是（接收方）如何在某个地点精确或近似地再现（发送方）在另一个地点传递的信息。

通用编码（universal code）
通过将每个整数映射为唯一的二进制编码来为正整数创建变长编码的一种方法。一般来说，最小的整数被赋予的二进制位数最少。

统计偏度（statistical skewing）
在概率论和统计学中，偏度是度量实值随机变量相对其均值而言概率分布的不对称性的。或者通俗地说，偏度就是度量某些符号比其他符号更可能出现的程度。例如，英语中各个字母的出现是偏度的，字母“e”比字母“q”更常见。对数据压缩来说，出现偏度是好事，一些数据转换操作符号集的目的就是为了增加偏度。

统计压缩（statistical compression）
根据输入符号的概率来确定输出符号的数据压缩技术。

详见第 5 章。

统一码（Unicode）
Unicode 计算机科学领域里的一项行业标准，用于对世界上大多数书写系统中的字符进行一致的编码、表示和处理。最新版本的 Unicode 中包含的字符超过 120 000 个，涵盖了 129 种现代的和历史的语言以及多个符号集。

唯一可译码（uniquely decodable codes）
如果一种编码的扩展是非奇异的，那么这种编码就是唯一可译的，这也就意味着目标符号是唯一可识别的。

无损数据压缩（lossless data compression）
即在压缩过程中不丢失任何信息的数据压缩技术，并且压缩后的数据可完整准确地还原为原始数据。

线性相关（linear correlation）
一种通过共现或变化模式观察到的两个事物之间的关系或联系，如果其中一个发生变化，则另一个也会相应地发生线性变化。例如，如果气温升高，那么冰淇淋的销量就会增加。最重要的是线性相关并不意味着存在因果关系。

香农 - 范诺编码（Shannon–Fano coding）
一种通过一组符号及其出现的概率（估算出来的或测量所得的）来构造**前缀编码**的技术。与**哈夫曼编码**一样，它不能实现尽可能短的码字长度预期，在这个意义上是次优的。但与哈夫曼编码不同的是，它能确保所有的码字长度与其理想的理论值之差控制在一个二进制位以内。

信息（information）
就我们的目的而言，是指消息（message）的内容。信息可以被编码成各种形式以供传输和解释（例如，信息可以被编码成符号序列或通过信号序列传输）。

信息可以消除不确定性，事件的不确定性是通过其发生的概率来衡量的，并且与发生的概率成反比。一个事件的不确定性越大，就需要越多的信息来消除其不确定性。

信息论（information theory）
数学、电子工程和计算机科学中涉及信息的量化（quantification of information）的分支学科。信息论研究信息的传递、处理、利用和提取。

信息内容（information content）
数据流中包含的实际信息（与**噪声**相对）。参见词条**信息熵**。

信息熵（information entropy）
在消息中存储或传输一个符号所需的平均位数。

详见第 3 章。

信源（sources）
能够对消息数据进行编码并通过信道向一个或多个信宿传输信息的实体。任何能够产生连续消息的过程都可称为信源，也可称为“发送者”（sender）。

序列化（serialization）
把对象或数据结构转换为可存储或可传输的位序列的过程。这意味着可以使用反序列化来重建原始对象。

一元编码（unary encoding）
一种熵编码，在这种编码中，自然数 n 有两种表示方法，一种是用 n 个 1 后面加一个 0 来表示（这里，**自然数**被理解为**非负整数**），另一种则是用 n–1 个 1 后面加一个 0 来表示（这里，**自然数**则被理解为**严格正整数**）。例如，5 可以表示为 111110 或者 11110。其中，0 和 1 是可以相互交换的。

依词典顺序排列（lexicographic permutation）
将字符串依词典顺序重新排列使得相同的字符聚在一起，参见词条**伯罗斯 – 惠勒变换**。

有损数据压缩（lossy data compression）
是指在压缩过程中会丢失一些信息的数据压缩技术，原始数据无法完整准确地还原。有损数据压缩的目标是最大程度地在压缩数据与对原始数据的足够保真之间寻求平衡。

有限状态熵（finite state entropy，FSE）
非对称数字系统的具体实现，其关注重点是提高性能。

噪声（noise）
任何对信源与信宿之间所传输的信号或信息的干扰都可称为噪声。

折纸数理学（mathematical origami）
从数学的角度研究折纸。

直方图（histogram）
由很多矩形组成的图，其中矩形的面积与变量的频率成正比，而其宽度则等于类间隔。

字典编码（dictionary encoding）
基于最常见的符号分组方法转换数据流的过程。

详见第 7 章。

字节码（bytecode）
由紧凑的数值代码组成的指令集，专为软件解释器高效执行而设计。

字面值词条（literal token）
一种输出词条，提示下一个符号应从字面值流中读取或写入字面值流。

详见第 6 章。

字面值流（literal stream）
只包含符号的字面（未编码）值的数据流。

详见第 6 章。

最低有效位（least significant bit，LSB）
二进制数中权值最小的位。例如，对二进制数 1000 来说，最右边的 0 就是最低有效位。

最高有效位（most significant bit，MSB）
二进制数中权值最高的位。例如，对二进制数 1000 来说，最左边的 1 就是最高有效位。

关于作者

柯尔特·麦克安利斯（Colt McAnlis）是一位谷歌开发倡导者，专注于游戏开发、压缩技术和性能提升。在此之前，他是一名游戏行业的图形程序员，曾任职于暴雪娱乐、微软全效工作室（Ensemble）和岩石壁画公司（PetroGlyph）[1]。此外，他还曾担任南卫理公会大学 Guildhall 学院的兼职教授，优达学城（Udacity）的讲师（两次），同时还是一位作者。最近，他一直在教授安卓开发人员程序性能提高之道。闲暇时，他将时间花在抵御外太空巨型蚂蚁的入侵上。他在网络上发布了大量的作品、视频和其他资源，总浏览量超过 60 万次。

亚历克斯·海奇（Aleks Haecky）同样来自谷歌，他是一位开发倡导者、培训开发人员和作者，致力于弥合专家与普通读者之间的语言鸿沟。他曾从事性能提升、文档编写等幕后工作，在优达学城、谷歌开发者频道也从事一些幕后工作。此外，他还翻译过爬行动物学相关的图书，并教人学习皮划艇。更不用说，他正在准备写下一部伟大的美国小说，并在 LinkedIn 上深度潜水。

关于封面

本书封面上是一只巴西三带犰狳。

顾名思义，这种犰狳为巴西所特有。它们主要生活在开阔的大草原和干燥的林地中，喜欢在较高的草丛、灌木丛和荆棘林中栖息。三带犰狳通常在夜间活动，但也会在白天觅食。它们主要以蚂蚁和白蚁为食，这些猎物是其用鼻子靠近地面闻出来的，它们能闻到 20 厘米深的土壤中隐藏的猎物气味。三带犰狳擅长挖地洞，但是它们更喜欢在灌木丛里而不是洞穴里休息。它们同样不依靠挖地洞来防守，而是滚成一个球，外面只剩下坚硬的护甲，是仅有的两种能缩成球形的犰狳之一。

犰狳通常是独居动物，但是三带犰狳偶尔也会以不超过 3 个成员的小家庭方式群居。每年的 10 月至来年的 1 月为交配期，在交配前它们会有短暂的求偶行为。犰狳的妊娠期一般为 120 天左右，每胎产 1 仔。刚出生的犰狳眼睛是闭着的，护甲很柔软，但是它的爪子已经发育成熟，可以在出生后几个小时内行走并滚动成一个球。在过去的 10 年里，巴西三带犰狳的数量减少了 30%。在自然界中它们仅有的天敌是成年美洲狮和美洲虎，但它们面临的主要威胁是栖息地的破坏，因为人类占领了它们的栖息地饲养牲畜。

很多 O'Reilly 的图书封面的动物濒临灭绝，但它们对世界很重要。想了解如何救助这些动物的更多信息，请访问 animals.oreilly.com。

本书封面图片选自 *Beeton's Dictionary*。

注 1：这三家公司推出的著名游戏分别为星际争霸与魔兽争霸、帝国时代和星球大战。——译者注



微信连接



回复“算法”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

数据压缩入门

5G时代即将到来，随之而来的是铺天盖地的数据洪流。对此，你和你的公司做好准备了吗？用户随时随地分享大量图片、音频和视频，应用程序的云端放得下吗？运营商的传输速度跟得上用户的分享速度吗？客户端能否快速加载用户点击的页面？

要在5G时代成功获取用户并提升转化率，离不开数据压缩的专业技能。本书从理论和实践两方面入手，面向开发人员讲解数据压缩算法，并帮助开发人员选择合适的数据压缩工具。书中通过讲解清晰、步骤详细的示例，将数据压缩算法化繁为简，帮助开发人员做出正确的有关数据压缩的商业决策，从而实现客户更多、事业更兴、利润更高。

- 了解5类数据压缩算法：变长编码、统计压缩、字典编码、上下文模型、多上下文模型
- 了解数据、场景和算法，以选择匹配的数据压缩工具
- 选择合适的图像压缩算法，权衡图像质量与文件大小
- 学习如何压缩客户端和服务器生成的数据
- 了解与数据压缩算法有关的名人及其趣事

“我希望本书能揭开数据压缩的神秘面纱，为软件开发人员学习压缩算法提供一个起点，同时帮助他们开发出更好的软件。”

——**John Brooks**
Blue Shift公司CTO

柯尔特·麦克安利斯（Colt McAnlis），谷歌开发倡导者，专注于游戏开发、压缩技术和性能提升。担任南卫理公会大学Guildhall学院的兼职教授，加州大学洛杉矶分校继续教育学院讲师，以及优达学城（Udacity）的讲师。

亚历克斯·海奇（Aleks Haecky），谷歌开发倡导者、培训开发人员，从事性能提升、文档编写等幕后工作，也在优达学城、谷歌开发者频道从事一些幕后工作。

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095183转600

分类建议　计算机/算法

人民邮电出版社网址：www.ptpress.com.cn



O'Reilly Media, Inc.授权人民邮电出版社出版



ISBN 978-7-115-53417-0
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks